SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2019;**29**:e1699.
Published online 14 June 2019 in Wiley Online Library (wileyonlinelibrary.com). https://doi.org/10.1002/stvr.1699

A formal approach to automatically analyse extra-functional properties in mobile applications

Ana Rosario Espada, Maria del Mar Gallardo, Alberto Salmerón, Laura Panizo and Pedro Merino*,†

Universidad de Málaga, Andalucía Tech, Dept. Lenguajes y Ciencias de la Computación, Málaga, Spain

SUMMARY

This paper presents an integrated approach for testing mobile applications (apps) against a set of extra-functional properties to be used by app developers. The approach starts with the (manual or automatic) extraction of the interaction model, that is, a formal model of the potential user interactions with the app. The model is constructed to allow a model checking tool to exhaustively extract the so-called app user flows, that is, the sequences of user actions, that constitute the test cases. In the final step, the app user flows are executed on the app running on real devices. The resulting execution traces are enriched with different measures and verified against a set of extra-functional properties of interest. The approach has been adapted to analyse several applications running at the same time with several devices supporting the applications. This paper presents the definition and formalization of both the modelling language for the interaction model and the specification language to represent the extra-functional properties. It also describes a methodology for automatically extracting the model. Finally, it presents an implementation focused on Android apps, which is integrated in the TRIANGLE testing framework, and the evaluation of the approach. © 2019 The Authors. *Software Testing, Verification & Reliability* Published by John Wiley & Sons Ltd.

Received 6 October 2016; Revised 19 March 2019; Accepted 26 March 2019

KEY WORDS: analysis of traces; android; extra-functional properties; model-based testing; verification

1. INTRODUCTION

The automated analysis of applications that run on smartphones is a hot topic due to the increasing role of these platforms as the main way for users to connect to the Internet. Execution errors and underperformance in mobile apps (the usual name for applications) have a great impact on user experience, on the overall behaviour of the smartphone and on the mobile communication network. This potential negative impact is not negligible considering that more than 2 billion devices run connected apps every day. In addition to the software's functional properties, the analysis of extra-functional properties (EFPs), such as energy consumption, response time, traffic generation and memory use, is a central issue in developing new techniques to verify mobile apps.

The current application of formal methods, such as variants of static analysis or model checking, does not effectively predict the behaviour of mobile apps regarding EFPs due to the difficulties in constructing a realistic model of the whole environment where the application is running, including user interaction, other apps running on the same device, the operating system and interaction with the mobile networks. Approaches that work with models such as App Explorer [1] or PerfChecker [2] still need more accurate information on delays or energy consumption, which can be

*Correspondence to: Pedro Merino, Dept. Lenguajes y Ciencias de la Computación, Universidad de Málaga, Andalucía Tech, Málaga 29071, Spain.

†E-mail: pedro@lcc.uma.es

obtained only from real executions. On the other hand, approaches that rely only on runtime monitoring to characterize app behaviour lack the mechanics to generate all realistic scenarios and/or to formally ensure the correctness or coverage of the analysis. For instance, tools such as AntMonitor [3], NetworkProfiler [4] and ProfileDroid [5] support methods to explore several executions to produce statistics, identify reference patterns in the traffic or locate potentially suspicious behaviours, but they lack a suitable formal framework to control the coverage of the executions and to describe the EFPs to be analysed.

This paper presents a new approach to assist mobile application developers to test apps in different network scenarios. The approach combines model-based testing [6] and runtime verification [7] techniques to verify whether a mobile app satisfies a given set of extra-functional properties. It is worth noting that model checking [8] is the underlying method that makes it possible to automate both the generation of test cases and the analysis of traces. Thus, if the EFPs are violated, it is possible to locate the execution traces of the app that causes the violation. This proposal has been integrated into the TRIANGLE testing framework [9], which provides a mobile network in a laboratory. In addition, the TRIANGLE framework provides application automation and monitoring functionality to allow the execution of tests in different controlled network scenarios.

Figure 1 shows a high-level overview of the testing framework, clearly separating the proposed methodology to describe both the interaction model and the expected properties (at the upper part of the figure) and the particular implementation of these ideas in TRIANGLE (the lower part of the figure). The app developer provides to the TRIANGLE Portal the binaries of the app (*APK* file in Android) and the extra-functional property (EFP), the interaction model and a description of the sequences of interactions suitable for the test cases (*app user flow requirements*). To ease the construction of the interaction model, the framework provides a support tool that extracts the interaction model from the application binaries. The interaction model and the app user flow requirements are then transformed and analysed with the SPIN model checker in order to produce a set of *app user flows*, that is, a set of realistic user actions suitable to evaluate the EFP. The *Experiment Control & Mobile Network Testbed* is in charge of executing the sequence of actions in a real device under different emulated network scenarios and providing the corresponding execution traces.

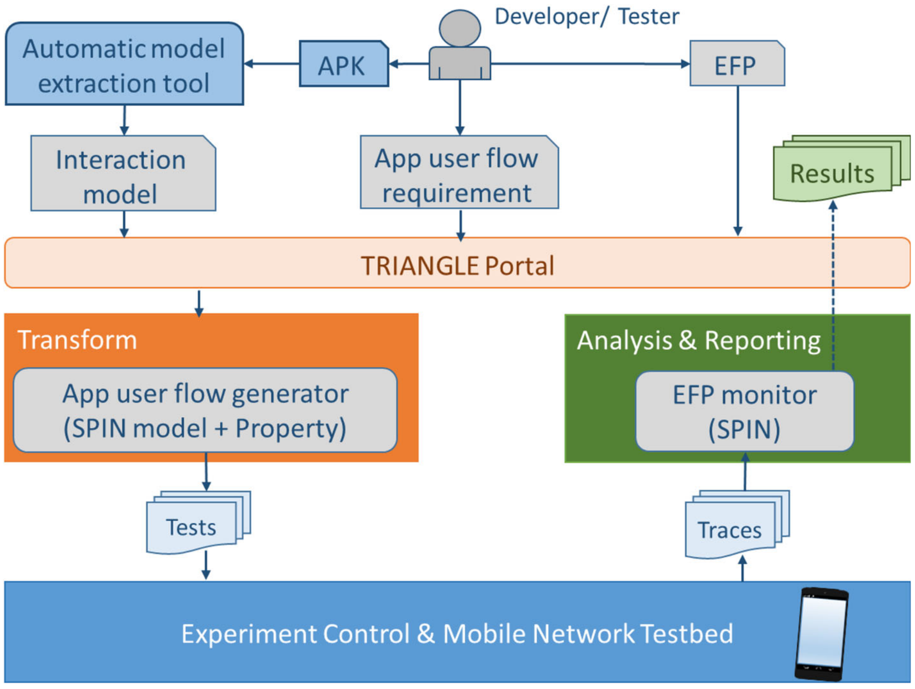
Developer/ Tester
Automatic model extraction tool
APK
EFP
Interaction model
App user flow requirement
Results
TRIANGLE Portal
Transform
App user flow generator (SPIN model + Property)
Analysis & Reporting
EFP monitor (SPIN)
Tests
Traces
Experiment Control & Mobile Network Testbed

Figure 1. Overview of the testing framework.

The verification of the EFP is performed by a monitor that compares events and states in the traces with respect to formal descriptions of the EFPs. The main contributions of this paper, as shown in Figure 1, are the following:

1. *A model-based method to generate test cases* for apps reacting to user interactions and internal events from the device (most likely from the network). Such external and internal interactions can be represented by a formal model (*interaction model*) with information provided by the app designer and/or by automating the model extraction (*automatic model extraction tool). The formal model is then explored with a model checker to produce realistic sequences of interactions with the running app. In addition, the method is optimized to generate only the sequences of interactions that are suitable for each test case (*app user flow requirement).
2. *A formal language to describe EFPs* related to the behaviour of apps in a given time slot, for which a set of interval-based logic formulae are defined to represent desired patterns for energy, traffic, memory, etc. with regard to time slots. Such basic patterns can be combined to describe complex behaviours.
3. *The formalization of the whole approach,* providing operational formal semantics for both the generation of test cases and the verification of the EFPs. Such semantics offer advantages such as the expansion to many interacting apps, internal events or time aspects.
4. *A testing workflow* that allows the integration of test case generation, execution of the apps in the device, control of additional monitoring hardware and verification of EFPs (using a model checker). This workflow allows a fully automated testing process.
5. *Implementation of the workflow* for Android devices as part of the TRIANGLE testing framework and validation with real applications.

Some of these contributions are new, while others are significant improvements of preliminary work by the authors [10–13], which proposed earlier versions of the ideas on automatic test case generation and the method to analyse energy-related properties. The current paper extends the initial method to generate test cases [10], where UML state machines were used as the modelling language for user interactions, with a precise definition of the variant of state machines that was finally implemented in the tool chain. In previous work [13], the authors proposed a preliminary method to guide the generation of app user flows by means of app user flow requirements. However, the interaction model was written by the user, a potentially tedious and error-prone process. The implementation described at present is completely new and is integrated into the TRIANGLE testing framework. The semantics of both modelling languages to describe user behaviours and EFPs were separately introduced in the previous papers, whereas this paper provides an updated and unified notation for both, making them easier to use as a reference for implementation. The authors used an earlier prototype to analyse traffic and energy in the well-known application Spotify for Android [12]. This paper presents the evaluation of the integrated approach using a new case study, likewise using a real app such as the Universal Music Player, and evaluates its behaviour in different network scenarios. In addition, the extended comparison with related work is completely new.

The remainder of this paper is organized as follows. Section 2 describes and formalizes the modelling language for the description of user behaviour. Section 3 introduces the language for the specification of EFPs and its transformation into temporal logic formulae to be analysed by a model checker. Section 4 introduces the TRIANGLE testing framework architecture and explains the contributions of this paper within this framework. Section 5 provides some details on the implementation. Section 6 presents the evaluation of the approach and some experimental results, and Section 7 reviews related work and highlights the contributions of this work compared to the state of the art. Finally, Section 8 summarizes the conclusions.

2. MODELLING EXPECTED USER BEHAVIOUR FOR TEST CASE GENERATION

Applications are defined by their behaviour. However, this behaviour is principally triggered through user interactions with the application's interface. Thus, this approach models applications from the user perspective, describing actions and sequences that make sense to the user. As mentioned earlier, the aim is to construct realistic test cases, that is, sequences of user actions that correspond to typical user behaviours. This section presents the concepts needed to model the interaction of the user and the mobile application, the modelling language and its formalization.

2.1. *Elements of mobile applications*

Users interact with a mobile application mainly through graphical elements called controls, for example, 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 gestures such as pinch to zoom. Not all controls respond to these gestures, for example, a button may react to taps but not to swipes. In addition, some user actions may also depend on events that they do not directly control. For instance, a 'play' button may be disabled while a song is being downloaded.

Due to the constraints of mobile devices and their small displays, most applications show only part of their graphical interface at a time. The set of controls that fits into the display at the same time comprises what is usually called a *screen*. While interacting with the application, the contents of the current screen may be replaced with others. For instance, an email application may have a screen with the list of emails in the inbox. When one of the emails is tapped, the application shows a different screen with the contents of the message.

Figure 2 shows three screens of the Universal Music Player app, which will be used as a case study throughout the paper. This music player is a sample app that is included in the Android Studio IDE and can also be obtained from the Google Play Store. Each screen shows different controls. For instance, initially, the app shows a list of songs (left screen). If a song name is clicked, then the app starts to play the song and shows some playback controls and the album cover image (central screen). In this state, if the album cover image is clicked, a completely new screen is shown with the complete playback controls (right screen). Observe that some interactions happen within the screen, such as tapping on a song to start playing, while others change the current screen, such as tapping on an album photo to open the full-screen player.

Navigation between screens presents interesting challenges. Many applications organize their screens hierarchically: new screens are higher in the hierarchy, and the user can also navigate back to the previous screen 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.

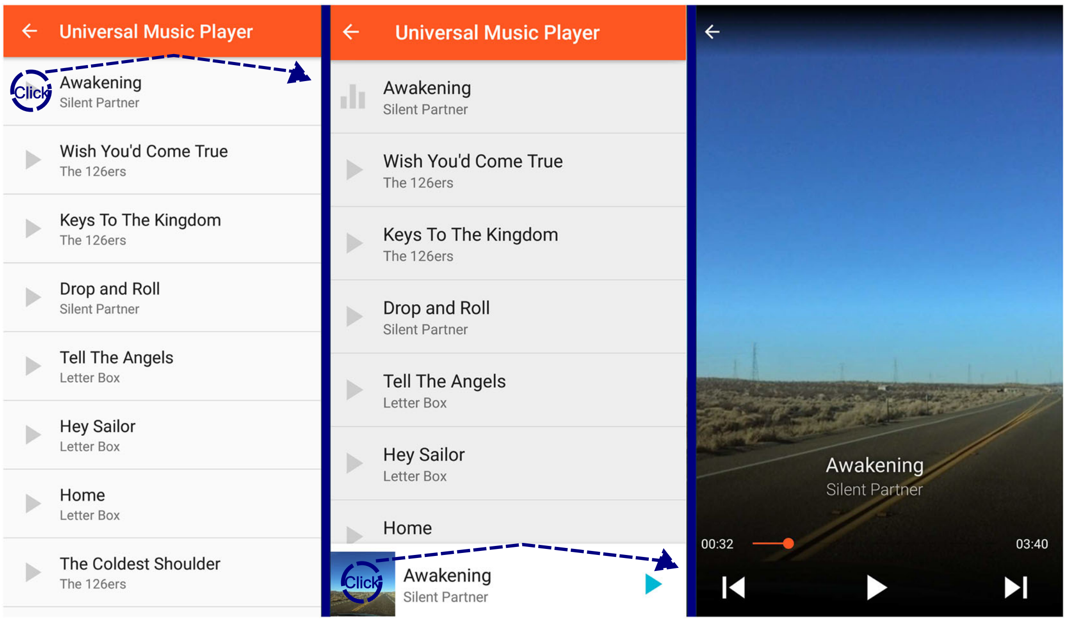

Figure 2. Universal Music Player – different UI states.

This navigation model can be pictured as a stack of screens, where the screen at the top of the stack is the screen 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 certain 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; for example, an email application could start the web browser application to load a website when a link is tapped. The new application might start at a screen other than its 'main' screen depending on the request made by the first application.

Not all behaviours in mobile applications can be described solely through user actions. Some depend at a certain point on events that are not directly controlled by the user, such an email being received or an alarm going off. Such events are called system events and must be taken into account in the proposed modelling language.

2.2. Modelling language

Given the elements of mobile applications identified in the previous section, a modelling language based on state machines is now proposed to describe them. This modelling language adopts many elements from UML state machines [14] and Harel statecharts [15], including their graphical notation, but is not a strict subset due to the introduction of additional elements to represent aspects of mobile applications.

State machines consist of states connected through labelled transitions. States are represented by rounded rectangles and transitions by directed arrows, both with an optional label. The unlabelled circles with two ongoing transitions are connection states, which will be explained later. State machines are also represented by rounded rectangles with a label, and they contain states and transitions. There are two special types of states: initial and final. The former are represented as unlabelled filled circles, while the latter are represented by circles with a cross inside.

Transitions represent user actions over the screen controls. Each transition is labelled with the action that the user would perform to progress to a new state. These actions include pressing a button, entering text into a field or scrolling up or down a list. Transitions may also be labelled with system events, which represent an event or condition not controlled by the user. This distinction is important for test case generation, as system events are not translated into actions performed on the screen. System events are distinguished by a '-' suffix in their names. Finally, the time elapsed between the current transition and the next can be specified by including the attribute '{time=x}' in the label.

Describing almost any application with a single state machine is impractical. Here, therefore, state machines are organized in two ways. First, a state machine may 'call' another state machine, for example, with each one representing the user behaviour on a different application screen. Second, state machines are contained in a hierarchy that mimics the elements identified earlier: devices, applications and screens. At the top level of the hierarchy are the *device state machines*. Each device contains one or more *application state machines*, which in turn may consist of one or more *activity state machines*. Activity state machines are useful for grouping different user behaviours in the same screen. At the bottom of the hierarchy are the state machines that define the different sets of user behaviours. They are called *user-view state machines* and contain states and transitions labelled with user/system events. This type of hierarchical composition is purely for convenience, as devices and user-view state machines would be logically sufficient.

Figure 3 shows part of the Universal Music Player model, which includes different nested state machines. 'UniversalMusicPlayer' is the application state machine. It contains three nested state machines that represent the app activities: 'MusicPlayerActivity', 'PlaceHolderActivity' and 'FullScreenActivity'. Similarly, these state machines include user-view state machines that specify the user's interaction with the app screen by means of states and transitions labelled with user or

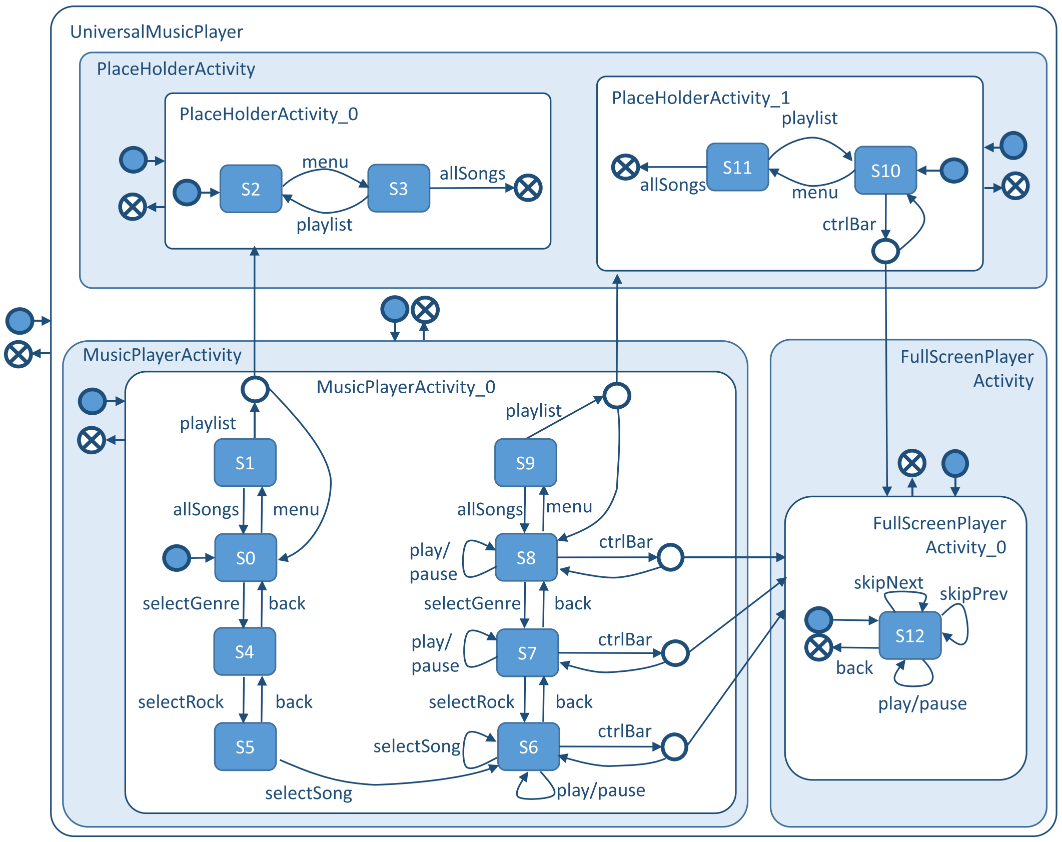

Figure 3. Universal Music Player interaction model.

system events. For instance, in 'MusicPlayerActivity_0', the transition from state *S4* to *S5* is fired when the user clicks on the play button, and the next transition takes place 300 s later.

A special type of state, called *connection state*, can be used to call another user-view state machine on the same device. Connection states are represented as unlabelled empty circles and have two outgoing unlabelled transitions: one to a target state machine and another to a returning state within the source state machine. When a connection state is reached, the execution continues with the user-view state machine referenced from the connection state. When the target state machine finishes, that is, when it reaches an end state, the execution resumes in the returning state.

Connection states can also reference an activity state machine. In this case, any *externally accessible* state machine within the activity can be executed next. External accessibility is a feature of applications, activities and user-view state machines and is represented with a pair of initial and final states reflecting the opportunity for access. The execution of the whole model starts with a user-view state machine that can be reached through a direct path of externally accessible elements, that is, applications and activities. This organization allows any state machine to be called explicitly from another.

In Figure 3, only the activity 'MusicPlayerActivity' is externally accessible, and thus when the application starts, only the state machine 'MusicPlayerActivity_0' is accessible. The other state machines are accessed through connection states. In this example, there is a connection state from state *S5* to the state machine 'FullScreenPlayerActivity_0', which is in a different activity. If the current state is *S5* and the user clicks on the control bar, the execution continues from the initial state of 'FullScreenPlayerActivity_0'. When this state machine reaches its final state, the connection activity indicates that the returning state is *S5*. From the *S5* state, the user can click the control bar again or click the pause button.

By following the available transitions from the initial state in the interaction model, an app user flow, that is, a sequence of user actions, is generated. If there is more than one available transition from a state, each one will lead to a different app user flow.

2.3. *Formal semantics of the modelling language*

In this section, the modelling language presented earlier and the method by which meaningful *app user flows* are constructed are formally defined. The use of mobile applications is formalized through the composition of *state machines* at three different abstraction levels. *User-view state machines* are at the lowest level and represent the user behaviour on a mobile screen. Users can interact with the active screen, firing user events. Sometimes one of these events activates a different screen. This control transfer between views is modelled through the *binary composition relation* between user-view state machines from which *device state machines* are constructed. Device state machines use *connection states* to switch from the current user-view state machine to a different one. This formalization does not take into account whether both user views belong to the same or different applications. Finally, the third level corresponds to the concurrent execution of device state machines. At this level, different mobile devices are assumed to be executing and interacting through some applications. Note that activity state machines, as described in the previous section, are only structural components of models that do not essentially affect the app description. Consequently, to simplify the formalization in the description hereafter, they have been omitted.

2.3.1. *User-view state machines.*

Definition 1
A *user-view state machine* is a labelled transition system $M = \langle \Sigma, I, \xrightarrow{-}, E, C, F \rangle$, where Σ is a finite set of states, $I \subseteq \Sigma$ are the initial states, $C \subseteq \Sigma$ are the so-called *connection states*, $F \subseteq \Sigma$ is the set of final states, E is the set of user/internal events and $\xrightarrow{-} \subseteq \Sigma \times E \times \Sigma$ is the labelled transition relation. Sets I, C and F are mutually disjoint.

Final states are states from which it is not possible to evolve. *Connection states* are states from which it is possible to transit to a different state machine. These states are essential to model the switching between typical views of smartphone devices.

Let $Flow(M)$ be the set $\{s_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} s_{n-1} \cdots | s_0 \in I, s_{n-1} \in F \cup C\}$ of all *finite* sequences of states allowed by M, starting at an initial state and ending at a final or connection state. Each element of $Flow(M)$ is named *flow*. The *length* of a flow is the number of its states.

Definition 2
Given a flow of length n, $\phi = s_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} s_{n-1} \in Flow(M)$, the test case, $test(\phi)$, determined by ϕ is defined as the sequence of events occurring in ϕ; that is, $test(\phi) = e_1 \cdot \cdots \cdot e_n$. The set of test cases allowed by M is defined as $TC(M) = \{test(\phi) | \phi \in Flow(M)\}$.

The set of events E is divided into two disjoint sets: the set of *user events*, denoted by E^+, with events such pressing a button, and the set of *system events*, denoted by E^-, which includes, for instance, system responses to user requests. In the following, e^+ and e^- represent user events and system events, respectively, and e alone refers to events of either type.

According to Definition 2, test cases are finite sequences of user and system events. For instance, sequence $e_1^+ \cdot e_2^+ \cdot e_3^- \cdot e_4^+$ represents a test case where the user first fires events e_1^+ and e_2^+, the system then fires e_3^- and finally the user fires e_4^+. Thus, user and system events are handled similarly during the generation of test cases. The difference between them is important when test cases are transformed into executable code, as described in Section 5. User events will be transformed into non-synchronized calls to methods that simulate the real occurrence of the event, while system events will correspond to calls to synchronized methods which wait for the arrival of the system event.

2.3.2. Composition of user-view state machines. This section will describe how user-view state machines are composed to construct flows that navigate through different user views. A *binary relation* $\mathcal{X}$, between connection and initial states, models this navigation. Assume that the flow in execution belongs to a user-view state machine M_i and that a connection state cs of M_i has been reached. If relation $\mathcal{X}$ defines a transition from cs to some initial state of another machine M_j, the flow could jump from M_i to M_j and then proceed following the transition relations of M_j. This jump implies a change in the active view from M_i to M_j. Subsequently, the user-view state machine that is visible in the device is called *active*, and the other user-view state machines, which have been created but are not currently visible in the device, are called *created*.

Assume a finite family of n state machines $M_i = \langle \Sigma_i, I_i, \overset{-}{\rightarrow}_i, E_i, C_i, F_i \rangle$ such that no state is shared between machines; that is, $\forall 1 \leq i, j \leq n.\, i \neq j$ implies $\Sigma_i \cap \Sigma_j = \emptyset$. Then, the machine $\bigcup_{i=1}^{n} M_i = \langle \Sigma, I, \overset{-}{\rightarrow}, E, C, F \rangle$ is constructed, where

- $\Sigma = \bigcup_{i=1}^{n} \Sigma_i, I = \bigcup_{i=1}^{n} I_i, E = \bigcup_{i=1}^{n} E_i, C = \bigcup_{i=1}^{n} C_i, F = \bigcup_{i=1}^{n} F_i$, and
- $s_i \overset{e}{\rightarrow} s'_i \iff \exists 1 \leq i \leq n. s_i \overset{e}{\rightarrow}_i s'_i$

In addition, $\mathcal{E} \subseteq E$ denotes the set of *call events* that provoke the switch between active user-view state machines.

Definition 3
The *connection* of a finite family of n user-view state machines $M_1, \cdots, M_n$ is given by a binary relation $\mathcal{X} \subseteq C \times \mathcal{E} \times I$, which enables the transition from connection states to initial states.

In the following, 3-tuples (s_i, e, s_j) of $\mathcal{X}$ are denoted as $s_i \overset{e}{\rightarrow}_c s_j$. Note that the source and target machines i and j could coincide.

When one view is left to continue in another view, the caller view usually stays active, which means that the execution could return to that view in the future. To account for this behaviour, each connection state $s \in C_i$ is assumed to have a related state $return(s) \in \Sigma_i$, which represents the state to be returned to when the caller view continues its execution.

When a new view is created, the call event could specify some parameters that determine how it must be started or finished. For example, if the view has already been created, the caller can choose whether to reuse the previously created view or create a new view. Additionally, when the newly created view has finished its execution, the caller view could automatically become active or not. The Boolean functions $reuse, auto\_return : \mathcal{E} \rightarrow \{false, true\}$ establish these parameters for the call events. Although there are other parameters that can be defined in the call events, these two are sufficient to describe the mobile behaviour.

The *device state machine* is now defined as the composition of the behaviour displayed by the user-view state machines along with the connection relation.

Definition 4
Given a finite family of n state machines $M_1, \cdots, M_n$ and a connection relation of $\mathcal{X}$, as defined earlier, the *device state machine* $\mathcal{D} = \bigcup_{i=1}^{n} M_i \bigoplus \mathcal{X}$ is defined as the state machine $\langle \Sigma^* \times \Sigma^* \times \mathcal{E}^*, I, \overset{-}{\rightarrow}_d, E, F \rangle$ where Σ^* and $\mathcal{E}^*$ are the set of finite sequences of states of Σ and the set of call events of $\mathcal{E}$, respectively.

The states of device state machines are termed *configurations*. A configuration is a 3-tuple $\langle sh, rh, eh \rangle$ in which sequence sh is the list of states $s_0 \cdot s_1 \cdots s_n$ that have been visited thus far in a flow, where s_n is the current state in the active user-view state machine. Sequence rh is the stack of states $r_1 \cdot r_2 \cdots r_m$ that constitutes *the history* of the view machines that have been created (and not yet destroyed) in the device but are not currently visible. Each state r_i of $r_1 \cdot r_2 \cdots r_m$ is the *return* state of a connection state of a user-view state machine that was previously active but became inactive when a transition from this view to another user-view machine took place. Notably, $s_0 \cdots s_n$

and $r_1 \cdots r_m$ both represent sequences of states, that is, subsets of Σ^*, and the naming is purely to distinguish between these two components of a configuration. Finally, $eh = e_1 \cdots e_m$ is *the history* of events that provoked a user-view switch in the current execution. Observe that stacks rh and eh have the same length. If e_i in eh is an event that fired a view switch, then r_i is the state to be returned to when this new state machine finishes. In the succeeding text, ϵ represents the *empty* state/event history and $eh \cdot e_m$ (similarly, $rh \cdot r_m$) denotes a non-empty stack with event e_m (state r_m) at the top.

The following rules define the relation $\xrightarrow{-}_d$ which is constructed from the transition relations of user-view state machines $\xrightarrow{-}_i$, and the binary connection relation $\xrightarrow{-}_c$. In these rules, given a history of states $r_1 \cdots r_m$ and an index j of a user-view state machine M_j, the function $top : \Sigma^* \times \mathbb{N} \to \Sigma \cup \{\perp\}$ returns the last state of the user-view state machine M_j in the sequence $r_1 \cdots r_m$. That is, $top(r_1 \cdots r_m, j)$ returns r_k, if $1 \le k \le m$ is the largest index such that $r_k \in \Sigma_j$, or $\perp$, if such a state does not exist.

$$\textbf{R1.}\ \frac{s \xrightarrow{e}_i s'}{\langle sh \cdot s, rh, eh\rangle \xrightarrow{e}_d \langle sh \cdot s \cdot s', rh, eh\rangle}$$

$$\textbf{R2.}\ \frac{s \xrightarrow{e}_c s', \neg reuse(e)}{\langle sh \cdot s, rh, eh\rangle \xrightarrow{e}_d \langle sh \cdot s \cdot s', rh \cdot return(s), eh \cdot e\rangle}$$

$$\textbf{R3.}\ \frac{s \xrightarrow{e}_c s', s' \in I_j, reuse(e), top(rh, j) = \perp}{\langle sh \cdot s, rh, eh\rangle \xrightarrow{e}_d \langle sh \cdot s \cdot s', rh \cdot return(s), eh \cdot e\rangle}$$

$$\textbf{R4.}\ \frac{s \xrightarrow{e}_c s', s' \in I_j, reuse(e), top(r_1 \cdots r_m, j) = r_k}{\langle sh \cdot s, r_1 \cdots r_m, e_1 \cdots e_m\rangle \xrightarrow{e}_d \langle sh \cdot s \cdot r_k, r_1 \cdots r_{k-1}, e_1 \cdots e_{k-1}\rangle}$$

$$\textbf{R5.}\ \frac{s \in F, auto\_return(e)}{\langle sh \cdot s, rh \cdot r_m, eh \cdot e\rangle \xrightarrow{-}_d \langle sh \cdot s \cdot r_m, rh, eh\rangle}$$

Rule **R1** states that a transition inside a user-view state machine M_i corresponds to a transition in the device state machine. The new state s' is added to the list of visited states $sh \cdot s$. Rules **R2** and **R3** model a transition from a machine to a new machine (M_j, for some index j) when both the new state s' and the event e are added to the view and event histories of the current system configuration. Rule **R2** is applied when event e does not involve reusing a previously created view ($reuse(e)$ is false), while **R3** applies when a view of M_j should have been reused ($reuse(e)$ is true) but the current view history does not contain one ($top(r_1 \cdots r_m, j) = \perp$). Rule **R4** defines a transition from a machine to M_j by reusing a previously created view of M_j ($reuse(e)$ is true) stored in rh ($top(r_1 \cdots r_m, j) = r_k$). Finally, **R5** defines the case when the flow of the currently active view has finished, and the execution must continue with the state r_m stored at the top of the view history $rh \cdot r_m$. Otherwise, that is, if $auto\_return(e)$ returns false, the current configuration $\langle sh, rh, eh\rangle$ cannot evolve. Note that, for simplicity, this model omits the case when the developer modifies the usual flow when finishing an activity: instead of returning to the previous activity in the stack ($auto\_return$ is $true$), the developer can insert a different activity to jump from the closing activity ($auto\_return$ is $false$).

Given a *device state machine* $\mathcal{D} = \bigcup_{i=1}^{n} M_i \bigoplus \mathcal{X}$, the *trace-based semantics* determined by $\mathcal{D}$ ($\mathcal{O}(\mathcal{D})$) is given by the set of finite/infinite sequences of configurations (flows) produced by the transition relation $\xrightarrow{-}_d$ from an initial state, in other words, $\mathcal{O}(\mathcal{D}) = \{\langle s_0, \epsilon, \epsilon\rangle \xrightarrow{e_0}_d \langle s_1 \cdot s_1, rh_1, eh_1\rangle \cdots | s_0 \in I\}$.

Definition 5

Given a flow $\phi = c_0 \xrightarrow{e_1}_d c_1 \xrightarrow{e_2}_d c_2 \cdots \in \mathcal{O}(\mathcal{D})$, the test case determined by ϕ is the sequence of events $test(\phi) = e_1 \cdot e_2 \cdots$. The set of *test cases* determined by a set of flows $\mathcal{T}$ is $TC(\mathcal{T}) = \{test(t) | t \in \mathcal{T}\}$.

Thus, a flow $\phi \in \mathcal{O}(\mathcal{D})$ consists of a sequence of user-view state machine flows (Definition 2) connected through *connection states*. Flow ϕ may finish at a final state of some user-view state machine or may be infinite. The *length* $|\phi|$ of a flow ϕ is the number of its states (configurations) if it is finite or ∞, otherwise. Given a flow $\phi = c_0 \xrightarrow{e_1}_d c_1 \xrightarrow{e_2}_d c_2 \cdots \in \mathcal{O}(\mathcal{D})$, the truncated flow up to length n, ϕ^n, is defined as ϕ iff $|\phi| \leq n$ or $\phi^n = c_0 \xrightarrow{e_1}_d c_1 \xrightarrow{e_2}_d c_2 \cdots \xrightarrow{e_{n-1}}_d c_{n-1}$, otherwise. To address this circumstance, the set of traces $\mathcal{O}^n(\mathcal{D})$ is defined as the set of all traces of $\mathcal{O}(\mathcal{D})$ that are truncated up to length n, in other words, $\mathcal{O}^n(\mathcal{D}) = \{\phi^n | \phi \in \mathcal{O}(\mathcal{D})\}$.

Observe that the state space of device state machines is not finite, given that the configurations include the state, view and event histories, which may have arbitrary lengths. In addition, the state space generated when an explicit model checker is constructing all the flows allowed by a device state machine is non-finite. This circumstance is due not only to the state and event histories, but also to the necessity for the matching algorithm, applied during the state space search, to account for both the current state of the flow and the history of the previous states of the flow, as contained in the first component of each configuration. For example, this approach allows both flows $\phi_1 = \langle s_0, \epsilon, \epsilon\rangle \xrightarrow{e_1^+}_d \langle s_0 \cdot s_1, \epsilon, \epsilon\rangle \xrightarrow{e_2^+}_d \langle s_0 \cdot s_1 \cdot s_2, \epsilon, \epsilon\rangle \xrightarrow{e_3^+}_d \langle s_0 \cdot s_1 \cdot s_2 \cdot s_3, \epsilon, \epsilon\rangle$ and $\phi_2 = \langle s_0, \epsilon, \epsilon\rangle \xrightarrow{e_4^+}_d \langle s_0 \cdot s_4, \epsilon, \epsilon\rangle \xrightarrow{e_1^+}_d \langle s_0 \cdot s_4 \cdot s_1, \epsilon, \epsilon\rangle \xrightarrow{e_2^+}_d \langle s_0 \cdot s_4 \cdot s_1 \cdot s_2, \epsilon, \epsilon\rangle \xrightarrow{e_3^+}_d \langle s_0 \cdot s_4 \cdot s_1 \cdot s_2 \cdot s_3, \epsilon, \epsilon\rangle$ to be generated by the model checker. Even though both visit the states s_1, s_2 and s_3, the path taken in each flow is different.

As a consequence, the models of device state machines are not, in general, state finite, which means that the model checking process does not, in general, terminate. The current implementation solves this problem by bounding the depth of the execution flows analysed by generating $\mathcal{O}^n(\mathcal{D})$ for some fixed n.

2.3.3. Composing several devices. The extension of the state machine model to several devices is carried out by composing the device state machines through interleaving. Thus, if $c_0 \xrightarrow{e_1}_d c_1$ and $c'_0 \xrightarrow{e'_1}_{d'} c'_1$ are transitions in devices $\mathcal{D}$ and $\mathcal{D}'$, respectively, then they allow the two transitions $\langle c_0, c'_0\rangle \xrightarrow{e_1}_{d||d'} \langle c_1, c'_0\rangle$ and $\langle c_0, c'_0\rangle \xrightarrow{e'_1}_{d||d'} \langle c_0, c'_1\rangle$ in the interleaved composition of $\mathcal{D}$ and $\mathcal{D}'$.

The communication between the two devices is modelled by a user event in the sender device (the device that starts the communication) and a system event in the receiver device (the device that receives the message). Thus, for instance, using the previous example, if $e_1 = e_1^+$ is an event that implies a communication from $\mathcal{D}$ to $\mathcal{D}'$ and $e'_1 = e_1^-$ is the corresponding event to be read by $\mathcal{D}'$ from $\mathcal{D}$, the test cases $e_1^+ \cdot e_1^-$ and $e_1^- \cdot e_1^+$ are generated. Note that in the second test case, the method that implements the transition for the receiver event will suspend the execution of $\mathcal{D}'$ until event e_1^+ is fired by $\mathcal{D}$.

In addition, when addressing the use of more than one device, model checking optimization techniques such as partial order reduction [8] are used to avoid the generation of multiple test cases that correspond to a single feasible interaction between the devices.

3. SPECIFICATION LANGUAGE FOR EXTRA-FUNCTIONAL PROPERTIES

This section introduces the language to describe EFPs on execution traces. Usually, verification techniques such as model checking evaluate properties over traces, abstracting the real time when each state occurs. This abstraction is adequate, for example, to analyse functional (safety and liveness) properties. However, the analysis of some non-functional properties, such as energy consumption, requires taking into account (tracking and measuring) the values of some non-discrete variables that *evolve over time*. For instance, to analyse the energy consumed by a device when downloading a file, it is necessary to determine when the download starts and finishes and measure the energy consumed by the device during this period.

Three interval formulae constitute the proposed simple specification language. 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 the traces. The interval formulae are first presented on an intuitive level and then formalized. Finally, this section addresses their translation into linear temporal logic (LTL) to be analysed by on-the-fly automata-based model checkers such as SPIN.

3.1. Interval formulae

Assume that $\mathcal{O}(P)$ is the set of execution traces determined by a transition system $P = \langle \Sigma, \overset{-}{\longmapsto}, \mathcal{L}, s_0 \rangle$. Traces are sequences of states of the form $\pi = s_0 \longmapsto s_1 \longmapsto \cdots$.[‡] This section uses the term *execution traces*, or simply *traces*, to denote the sequences of states produced by a transition system since this generality suffices to describe the syntax and semantics of the interval formulae. In any case, in the following sections, the execution traces will correspond to the real sequences of states generated when the test cases are executed on the mobile devices. Let $\mathcal{F}$ be a set of state formulae to be evaluated on the states of Σ. The relation $\models \subseteq \Sigma \times \mathcal{F}$ associates each state with the state formulae that it satisfies, in other words, given $s \in \Sigma$, and $p \in \mathcal{F}$, $s \models p$ iff state s satisfies formula p. As usual, the state formulae are assumed to be constructed from a set of atomic propositions and Boolean operators.

Assume a set of variables $\mathcal{C}$ that represent continuous magnitudes to be analysed on the executions of traces. Each $c \in \mathcal{C}$ is a real-valued function $c : \mathbb{R}_{\geq 0} \to \mathbb{R}$ that defines the evolution of c with time. Thus, $c(t) \in \mathbb{R}$ gives the value of c at time instant t. State formulae are used to determine the state intervals on which it is necessary to measure the continuous variables. For example, the state formulae *wifi_on*, *wifi_off* may serve to detect the states in the traces during which the *wifi* is activated.

Following this idea, a simple language is proposed for the specification of non-functional properties, which will use the intervals of states to describe the periods during which the continuous variables of interest are evolving and must be monitored.

Given $p, q \in \mathcal{F}$, $c \in \mathcal{C}$ and $K \in \mathbb{R}_{\geq 0}$, three valid formulae are defined in the specification language: $[[\mathit{diff\_c} \leq K]]_{[p,q]}$, $\forall[[\mathit{diff\_c} \leq K]]_{[p,q]}$ and $\exists[[\mathit{diff\_c} \leq K]]_{[p,q]}$. The intuitive semantics of $\forall[[\mathit{diff\_c} \leq K]]_{[p,q]}$ is as follows. An execution of trace $\pi = s_0 \longmapsto \cdots$ satisfies $\forall[[\mathit{diff\_c} \leq K]]_{[p,q]}$ iff for each pair of states s_i, s_j of π, with $i \leq j$ and $s_i \models p, s_j \models q$, the following condition (Cond) holds: '*the difference between the values of variable c at the time instants when s_i and s_j took place is less than or equal to K*'. That is, p and q determine the trace interval $s_i \longmapsto \cdots \longmapsto s_j$ on which the evolution of variable c must to be observed. The other two formulae are similar, except that $\exists[[\mathit{diff\_c} \leq K]]_{[p,q]}$/$[[\mathit{diff\_c} \leq K]]_{[p,q]}$ holds iff condition Cond holds for some/the first pair of states s_i, s_j with $i \leq j$, and $s_i \models p, s_j \models q$.

For example, if variable c gives the energy consumed by a device, formula $\forall[[\mathit{diff\_c} \leq k]]_{[\mathit{wifi\_on},\mathit{wifi\_off}]}$ holds for an execution of a trace π iff each time the *wifi* is activated, the energy consumed is less than or equal to K.

3.2. Interval formulae semantics

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

Although the execution time is abstracted in operational semantics, it is clear that the execution of each trace takes time, during which many other things that influence or are affected by the trace execution may occur. The following definition makes the time taken by the execution of the traces explicit.

[‡]Because transition labels are not necessary, they are omitted from the transition relation.

[§]To simplify the presentation, it is assumed that traces have at least one non-ending state s_0.

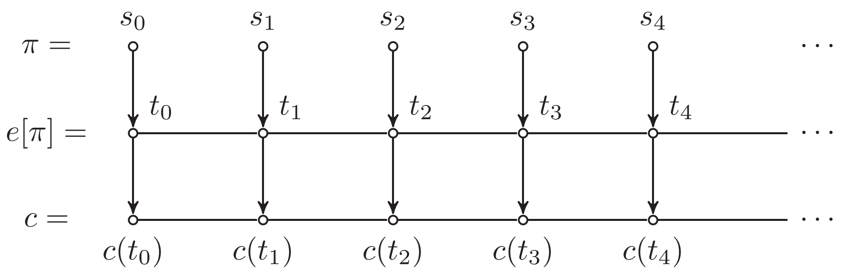

Figure 4. Synchronization of trace π and continuous variable c using $e[\pi]$.

Definition 6
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 relating states to different time instants. Once the execution of a trace $e[\pi]$ has been fixed, the values of the continuous magnitudes that evolve synchronously with the trace can be observed. The diagram in Figure 4 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 the continuous variable c in time and its synchronization with $e[\pi]$.

The intervals of states (within the traces) are used to determine the periods of time during which continuous variables should be observed. To accomplish this task, the interval calculus introduced by Chaochen and Hansen [16] is used to give formal semantics to the language for EFPs presented in Section 3.1. The interval logic domain 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 $diff\_c : \mathbb{I}ntv \to \mathbb{R}$ given as $diff\_c([t_1, t_2]) = c(t_2) - c(t_1)$.

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

Given a trace π and an interval of natural numbers $[i, j]$, $\pi \downarrow [i, j]$ denotes 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 to evaluate interval expressions such as $diff\_c \leq K$.

State formulae are used to construct state intervals as follows. Given the set of state formulae $\mathcal{F}$ defined in Section 3.1, the term *formula intervals* is applied to expressions such as $[p, q]$ with $p, q \in Prop$. The satisfaction relation $\models$ on state intervals is extended as follows.

Definition 7
Given a trace π and an interval of natural numbers $I = [i, j]$ with $i \leq j$, 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. $\pi(j) \models q$
3. $\forall i < k < j . \pi(k) \not\models q$

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.

[¶]It is assumed that $e[\pi](i)$ represents the time instant when state s_i is created.
[‖]It is assumed that if π is a trace of length n, $e[\pi]$ associates the final ending states of π with the time instant when the last non-ending state took place, that is, $\forall k \geq n . e[\pi](k) = e[\pi](n-1)$.

The following assumes that the ending state o satisfies no formula of $\mathcal{F}$, that is, $\forall p \in \mathcal{F}.\, o \not\models p$.

Now, given a trace π and a proposition interval $[p, q]$, $\pi \Downarrow [p, q]$ denotes 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 described earlier, that is, $\forall 0 \leq i < m.\, \pi \downarrow I_i \models [p, q]$.

The following two definitions show how the sequence of intervals $\pi \Downarrow [p, q]$ may be constructed.

Definition 8

Given a state formula p, 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 is ∞, otherwise. Then, $\pi \downarrow_k p$ can be inductively defined as:

1. $\pi \downarrow_k p = k$, iff $\pi(k) \models p$
2. $\pi \downarrow_k p = \pi \downarrow_{k+1} p$ iff $k < n, \pi(k) \not\models p$
3. $\pi \downarrow_k p = \infty$ iff $k \geq n$

Definition 9

Given a finite trace π and two state formulae p, q, the sequence of state intervals determined by p, q, $\pi \Downarrow [p, q]$, is inductively defined from operator $\Downarrow_k$ with $k \geq 0$, as described in the succeeding text.

1. $\pi \Downarrow_k [p, q] = \epsilon \iff \pi \downarrow_k p = \infty,\ or\ \pi \downarrow_k p = j \wedge \pi \downarrow_{j+1} q = \infty$ [**]
2. $\pi \Downarrow_k [p, q] = [j, l] \cdot (\pi \Downarrow_{l+1} [p, q]) \iff \pi \downarrow_k p = j \wedge \pi \downarrow_{j+1} q = l$

Then, $\pi \Downarrow [p, q]$ is defined as $\pi \Downarrow_0 [p, q]$.

Thus, the two state formulae $p, q \in \mathcal{F}$ determine a sequence of state intervals $\pi \Downarrow [p, q] = I_0 \cdots I_{m-1}$ in π that satisfy $[p, q]$. This definition can be extended to executions e of π as $e[\pi] \Downarrow [p, q] = e[\pi] \downarrow I_0 \cdots e[\pi] \downarrow I_{m-1}$. This approach provides semantics for the three formulae presented in Section 3.1.

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

Definition 10

Let Φ and $[p, q]$ be an interval expression and a formula interval, respectively. Let e be an execution of a finite trace π. Then,

1. $e[\pi]$ satisfies Φ on the time intervals determined by $[p, q]$, written as $e[\pi] \models [[\Phi]]_{[p,q]}$, iff $e[\pi] \Downarrow [p, q] = T_1 \cdots T_m (m > 0)$ and $\Phi(T_1)$ holds.
2. $e[\pi]$ satisfies $\exists\Phi$ on the time intervals determined by $[p, q]$, written as $e[\pi] \models \exists[[\Phi]]_{[p,q]}$, iff $e[\pi] \Downarrow [p, q] = T_1 \cdots T_m (m > 0)$ and $\exists 1 \leq i \leq m. \Phi(T_i)$ holds.
3. $e[\pi]$ satisfies $\forall\Phi$ on the time intervals determined by $[p, q]$, written as $e[\pi] \models \forall[[\Phi]]_{[p,q]}$, iff $e[\pi] \Downarrow [p, q] = T_1 \cdots T_m (m \geq 0)$, if $\forall 1 \leq i \leq m. \Phi(T_i)$ 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) $\exists[[\Phi]]_{[p,q]}$ iff a time interval exists in the sequence $e[\pi] \Downarrow [p, q]$ that satisfies Φ; (3) $\forall[[\Phi]]_{[p,q]}$ iff all the time intervals determined by $\pi \Downarrow [p, q]$ and e satisfy Φ. For instance, if $\Phi = diff\_c \leq K$, then $[[\Phi]]_{[swifi,ewifi]}$ establishes that the time interval determined by the first state interval on which $[swifi, ewifi]$ holds must satisfy Φ.

3.3. From interval properties to LTL

This section discusses how the interval properties can be practically evaluated on execution traces. Each type of interval property can be described by an LTL formula to be given to the model checker. To simplify the presentation of the formulae, given two state formulae p and q, $\Phi(p, q)$ is defined as:

$$\Phi(p, q) \equiv p \wedge (\neg q\, U\, (q \wedge \Phi))$$

[**] ϵ represents the empty sequence.

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'*, as the following diagram illustrates:

$$\begin{tikzcd}
\overset{p \wedge \neg q}{\underset{s_i}{\circ}} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{q \wedge \Phi}{\underset{s_j}{\circ}}
\end{tikzcd}$$

For $[[\Phi]]_{[p,q]}$ properties, the following LTL specification is used:

$$[[\Phi]]_{[p,q]} \equiv (\neg p)\ U\ \Phi(p, q) \tag{1}$$

The intended meaning of this formula is as follows. First, search for the first state (s_i) on which p holds; then, 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, if it is not possible to find either s_i or s_j, or if the time interval does not satisfy Φ, then the formula is false. The following sequence shows a trace that satisfies $[[\Phi]]_{[p,q]}$. Note that solid arrows are used to identify intervals and dashed arrows otherwise.

$$\pi = \begin{tikzcd}
\overset{\neg p}{\circ} \arrow[r, dashed] & \overset{\neg p}{\circ} \arrow[r, dashed] & \overset{\neg p}{\circ} \arrow[r, dashed] & \overset{p \wedge \neg q}{\underset{s_i}{\circ}} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{q \wedge \Phi}{\underset{s_j}{\circ}} \arrow[r, dashed] & \circ
\end{tikzcd}$$

For the property $\exists\, [[\Phi]]_{[p,q]}$, following LTL specification is used:

$$\exists\, [[\Phi]]_{[p,q]} \equiv \Phi(p, q)\ \vee\ \Diamond(\neg p \wedge \circ\, \Phi(p, q)) \tag{2}$$

that is, $\Phi(p, q)$ should be true either in the first state or in some future instant. Note that $\circ$ represents the 'next' operator, which is safe to use in this case because the analysing addresses linear execution traces rather than concurrent programs. The use of $\circ$ is needed to ensure that the formula $\Phi(p, q)$ is evaluated on a maximal time interval determined by p and q; that is, the state on which p is true must be preceded, if it is not the initial state, by a state that does not satisfy p. The following sequence shows an example of a trace for which $\Phi(p, q)$ holds on the second time interval determined by p and q.

$$\pi = \begin{tikzcd}
\overset{\neg p}{\circ} \arrow[r, dashed] & \overset{p \wedge \neg q}{\circ} \arrow[r, dashed] & \overset{\neg q}{\circ} \arrow[r, dashed] & \overset{q \wedge \neg \Phi}{\circ} \arrow[r, dashed] & \circ \arrow[r, dashed] & \overset{p \wedge \neg q}{\underset{s_i}{\circ}} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{q \wedge \Phi}{\underset{s_j}{\circ}} \arrow[r, dashed] & \circ
\end{tikzcd}$$

Finally, the LTL formula for $\forall\, [[\Phi]]_{[p,q]}$ is given by:

$$\forall\, [[\Phi]]_{[p,q]} \equiv p \to \Phi(p, q)\ \wedge\ \Box((\neg p \wedge \circ p) \to \circ\, \Phi(p, q)) \tag{3}$$

that is, all maximal intervals determined by $[p, q]$ properties in their ending points should satisfy Φ at the instant when the right ending point occurs. The following sequence shows a trace with two time intervals, determined by p and q, for which Φ is true. Note that the last state is labelled with the symbol o to indicate that the trace does not contain any state interval after $[s_{i_2}, s_{j_2}]$ satisfying $[p, q]$.

$$\pi = \begin{tikzcd}
\overset{\neg p}{\circ} \arrow[r, dashed] & \overset{p \wedge \neg q}{\underset{s_{i_1}}{\circ}} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{q \wedge \Phi}{\underset{s_{j_1}}{\circ}} \arrow[r, dashed] & \overset{p \wedge \neg q}{\underset{s_{i_2}}{\circ}} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{\neg q}{\circ} \arrow[r] & \overset{q \wedge \Phi}{\underset{s_{j_2}}{\circ}} \arrow[r, dashed] & \underset{o}{\circ}
\end{tikzcd}$$

4. TRIANGLE TESTING FRAMEWORK

The goal of the TRIANGLE project[††] is to provide app developers and device makers with benchmarking services to ensure that their products are ready to meet the challenges presented by 5G mobile networks. The improvements resulting from the upcoming standard will push existing use

[††]www.triangle-project.eu.

cases and enable new ones, for example, in connection with eHealth, smart cities and media streaming, and will increase the expectations of the end users. In this context, the TRIANGLE project focuses less on functional testing, which is covered by many existing tools and services, than on quality of experience (QoE). The benchmarking process is built on a set of key performance indicators (KPIs), which are organized around domains such as power consumption, network resource usage and user experience. Each KPI will provide insight into a different aspect of the application or device that influences user perception. The aggregation of the KPIs from each domain will be compared against a set of reference apps or devices.

Figure 5 presents a high-level overview of the TRIANGLE platform. The project involves building a testing framework using state-of-the-art testing equipment, which enables the emulation of many different network scenarios in a realistic manner, using real mobile devices and apps. This equipment is expensive for most developers, and its configuration and use is a difficult, error-prone task that requires a high degree of expertise. In addition, the framework consists of many pieces that must be coordinated to conduct a test. The project aims to hide all this complexity behind a user-friendly web interface allowing users to focus on their products and the results of the benchmarking. Users will see only high-level network scenarios, such as urban pedestrian or high-speed train environments, instead of the actual configuration of the emulation equipment. The user interacts with a web portal, where the app under test must be provided along with additional relevant information, such as the features supported by the app. The framework uses this information to decide which test cases are applicable to the app and to prepare and execute them. The execution of a test case involves coordinating the configuration of the network equipment, the automation of the user equipment (UE), that is, the mobile device, the measurement equipment and tools and the automation of the app itself. The processed results are shown to the user in the web portal.

This paper extends the original architecture of the TRIANGLE testing framework. First, the *Transform block* is extended by a formal definition of the interaction model that describes the user interactions with the app. A support tool has been implemented that automatically extracts the model from the compiled applications and allows the refinement of the model. Given the interaction model, model checking (exhaustive exploration) is used to obtain a set of app user flows that can be included

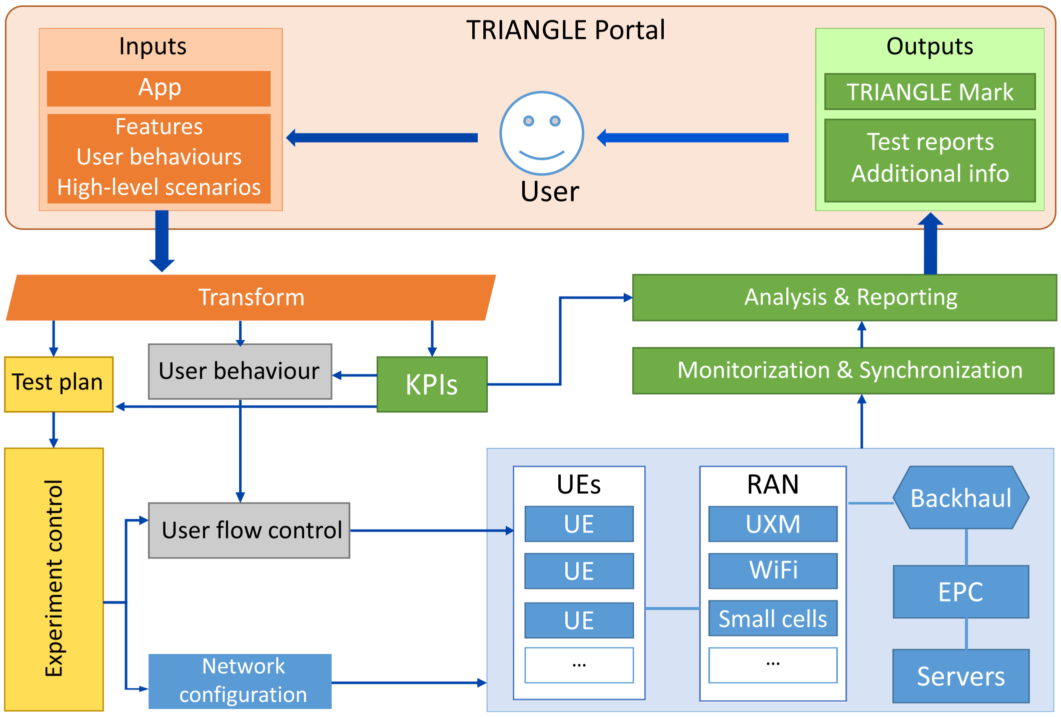

Figure 5. Overview of the TRIANGLE testing framework.

among the test cases to measure KPIs. Second, the *Analysis and Reporting block* is extended by means of a formal language to define EFPs. It is now possible to verify whether the traces produced during test case execution satisfy some of the properties.

4.1. *App automation*

One vital piece of the framework for benchmarking apps is guiding their execution. For instance, to execute a test in which the performance of a video playback will be evaluated, the app must be automated to play a video. The TRIANGLE framework uses Quamotion WebDriver [17], an automation solution for mobile apps based on the WebDriver standard for web browser automation. This tool can send commands to the mobile device that simulate actual user interactions with an app, such as tapping a button or entering text in a field. The sequence of actions used to carry out a test case in the TRIANGLE project is called the *app user flow*. The framework can take a JSON [18] script with a sequence of app user flows and perform them on the device using the Quamotion WebDriver.

Model-based testing [12] can help to automate the generation of app user flows as JSON scripts. Furthermore, if the model is correctly annotated, only the app user flows that are useful to compute any given KPI are generated (preliminary work [13]). If the requirements to measure a KPI change, then new app user flows can be generated from the same model. Therefore, the web portal can accept a single interaction model instead of a set of automation scripts. Section 5.1 presents an automatic way of creating this interaction model from the app itself so that most of the work is performed for the user. This process is based on the compiled version of the app (apk in Android). The generated model should still be fine-tuned by the app developer, for example, to add meaningful event names, but the bulk of the work will be performed automatically. The following sections presents the formal languages for modelling the app and describing the EFPs.

5. IMPLEMENTATION FOR ANDROID DEVICES

This section describes the implementation of this proposal for analysing mobile applications in the context of the TRIANGLE platform. Although this architecture can be adapted for different mobile operating systems, this paper focuses on its implementation for Android devices.

As discussed earlier in Sections 2 and 3, model checking, in particular the SPIN model checker, is the underlying formal technique (Figure 1), not only to generate app user flows that will be executed on real devices as part of the test cases but also to carry out the effective verification of EFPs, such as energy consumption, on these test cases.

5.1. *Model extraction*

Figure 6 shows an overview of the approach to automatically extract the interaction model. It is based on three main elements: the app controller, the exploration algorithm and the model parser. Given the application binaries, the app controller installs and manages the execution of the target app. The app controller can perform user events and capture the hierarchy of visual elements. The exploration algorithm decides the order in which the user events are performed and determines the different interaction model states based on the app UI. Finally, the model parser transforms the states and transitions obtained from the exploration into the interaction model. The following sections explain each element in more detail.

5.1.1. App controller. The app controller is in charge of interacting with the device where the application is running and is responsible for the following tasks:

1. Install, launch and close the application
2. Obtain the hierarchy of visual elements of the active view
3. Determine the list of visual elements that accept user events
4. Perform user events on specific visual elements (e.g. click, long click or scroll)
5. Fire system events (e.g. open/close keyboard or play/pause media)

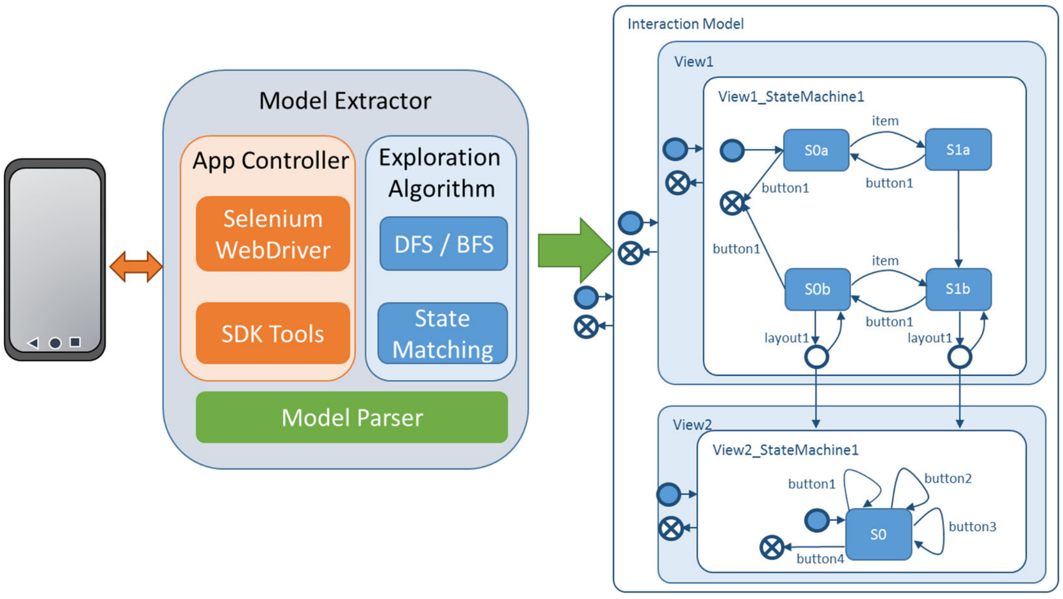

Figure 6. Automatic model extraction tool architecture.

Some of these tasks can be carried out with existing UI automation testing tools [19–22]. This approach has implemented an app controller for Android devices based on two testing frameworks. First, *Quamotion WebDriver* [17] is a test automation framework that automates iOS and Android apps on real devices. WebDriver [23] is an open protocol for test automation originally designed for web applications based on the exchange of JSON messages. Second, *Android UIAutomator* [22] is a UI testing framework included in the Android SDK. All previous tasks can be implemented with the SDK testing framework, but such implementation is less reusable. For this reason, UIAutomator is used only for tasks that WebDriver cannot carry out or cannot perform efficiently. In particular, UIAutomator's main task is to extract the app's Document Object Model (DOM), that is, the hierarchy of visual elements. For each visible element, the DOM includes a list of attributes: the resource identifier, the class and the user events accepted. This information is especially useful for determining whether the UI has changed after performing a user event and obtaining the list of controls and events that must be explored.

Mobile applications have complex UIs with multiple activities, fragments, overlays, controls, etc. that accept different types of user events. The exhaustive exploration of all visual elements that react to user events can lead, in the worst case, to large models with a poor performance in the generation phase of the app user flow. To manage the size of the application model, the app controller includes the following configurable options:

- Types of events considered in the exploration. For instance, if scrolling a layout produces the same effect as clicking on a tab menu, the scrollable views can be ignored and only click events analysed.
- Number of list items explored. In some apps, the effect of performing an event on any list item is the same. For instance, in a music player with a list of playable songs, clicking on each song will start playback. Thus, exploring only some of the items is sufficient to extract the model.
- Ignored elements. Some events in specific elements could have non-desired effects during the model extraction or the test execution, such as elements that restore the account password or open the configuration settings. For this reason, such elements are systematically excluded from the exploration.
- Predefined text for specific EditText fields. This concern is relevant in apps whose behaviour depends on the information provided in a form, for instance apps that require a login.

```
AppModel depthFirstSearch(Configuration s0) throws Exception{
 Set <Configuration> visited = new Set<Configuration>();
 Stack <Configuration> unvisited = new Stack<Configuration>();
 List <Configuration> path = new List<Configuration>();
 AppModel model;
 Configuration s;

 path.add(s0);
 unvisited.add(s0);

 while (! unvisited.isEmpty())
 {
   s = unvisited.pop();
   s.dom = performUserEvent(s.xPath, s.event);
   if ((v = visited.findByDom(s.dom))!=null)
   {
     if((v.xPath != s.xPath) && (s.event != n.event))
     {
       model.addTransition(path.getLast(),v, new
             Transition(s.xPath, s.event)));
     }
     backtracking(s,path);
   }else
   {
     model.addState(s);
     model.addTransition(path.getLast(),s,
           new Transition(s.xPath, s.event));
     visited.add(s);
     List<Configuration> successors = s.getSuccessors();
     if(successors!=null)
     {
       unvisited.addAll(successors);
       path.add(s);
     }else
     {
       backtracking(s,path);
     }
   }
 }
 return model;
}
```

Listing 1. Exploration algorithm based on DFS

Observe that most of the configurable options are connected to obtaining the list of visual elements that will be explored. Thus, the exploration algorithm examines a reduced number of states. In contrast, the configuration of EditText fields increases the number of explored states to include all desirable app behaviours.

5.1.2. Exploration algorithm. The exploration algorithm defines a *strategy* to execute user events in order to traverse the different app states. Amalfitano et al. [24] compared testing techniques and tools for mobile apps and determined that the most widely used strategies are depth-first and breath-first search. In the present work, both strategies were implemented, but only the results of the depth-first search are used in the evaluation section because the number of times that the app has to be initialized (or relaunched) is lower, which in turn decreases time required to extract the model.

An app state represents the app UI after the performance of a user event on a specific element. Because a state can be reached by performing user events in different UI elements, the exploration algorithm stores a 4-tuple (`xPath`, `event`, `dom`, `exp`), where `xPath` [25] identifies the element in the source DOM, `event` is the event performed on the element, and `dom` is the DOM after the event, in other words, the app state, and `exp` is a flag that, if true, indicates that the current configuration and its successors have been explored. A configuration has successors if the `dom` has controls that can handle user events. In the following, the term *configurations* is used for the 4-tuples handled by the algorithm to distinguish them from the interaction model states.

Listing 1 shows a simplified version of the algorithm in Java, which implements a depth-first search strategy. The main data structures are the set of `visited` configurations, the `path` (list) of configurations that leads to the current configuration and the stack of `unvisited` configurations. This stack stores incomplete configurations in the sense that the `dom` is empty because the event has not yet been performed. The input of the algorithm is the initial configuration `s0`, which is stored in `unvisited` with `xPath` '/', `event` 'launch' and an empty `dom` (line 9). While there are unexplored configurations, the algorithm extracts one, requests that the app controller perform the user event `s.event` on the visual element `s.xpath` (line 14) and stores the resulting `dom`. Then, the algorithm checks whether a configuration in `visited` has an equivalent `dom` (line 15). If so, the configuration is considered visited, and the interaction model is updated with a new transition. Otherwise, the configuration `s` is new and is included in `visited`. In addition, the interaction model is updated with the new state and the transition. If the configuration has successors, that is, if the `dom` has controls that can handle user events, they are included in the stack of unvisited configurations (line 32). Finally, the algorithm backtracks if the configuration `s` is in `visited` (line 22) or if `s` has no successors (line 36).

5.1.2.1. Matching criterion. The matching criterion defines when two `dom`s can be considered equivalent to diminishing the number of configurations explored. In Listing 1, the matching criterion is applied to determine if there is a configuration equivalent to `s` in `visited` (line 15). Listing 2 shows part of a `dom` provided by the app controller. Observe that it includes the hierarchy of visual elements (node) and information about them (user events enabled, class or text content).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<hierarchy rotation="0">
 <!-- ... -->
    <node index="0" text=""
        resource-id="com.example.android.uamp:id/toolbar"
        class="android.view.View" package="com.example.android.uamp"
        content-desc="" checkable="false" checked="false" clickable="false"
        enabled="true" focusable="false" focused="false" scrollable="false"
        long-clickable="false" password="false" selected="false"
        bounds="[0,75][1080,243]">
      <node index="0" text="" resource-id=""
          class="android.widget.ImageButton"
          package="com.example.android.uamp" content-desc="Open the main
          menu" checkable="false" checked="false" clickable="true"
          enabled="true" focusable="true" focused="false"
          scrollable="false" long-clickable="false" password="false"
          selected="false" bounds="[0,75][168,243]" />
      <node index="1" text="Universal Music Player" resource-id=""
          class="android.widget.TextView"
          package="com.example.android.uamp" content-desc=""
          checkable="false" checked="false" clickable="false"
          enabled="true" focusable="false" focused="false"
          scrollable="false" long-clickable="false" password="false"
          selected="false" bounds="[180,118][788,199]" />
    </node>
  <!-- ... -->
</hierarchy>
```

Listing 2. Example of DOM

The matching criterion determines that two `dom`s are equivalent if they are in the same activity, they have the same hierarchy (hierarchy relation and number of nodes), and their *nodes are equivalent*:

- Two nodes are equivalent if the following attributes are equal: resource-id, class, package, checkable, clickable, enabled, scrollable and long-clickable.
- If the nodes are editable, their text attributes must also be equal. A node is editable if its class is android.widget.EditText or inherits from this class. This rule is required to generate models of forms.

Figure 2 shows three different states of the Universal Music Player app. Applying the matching criterion, (*a*) (list of songs) and (*b*) (list of songs with first song playing) are different states; although they are in the same activity, they have a different number of nodes (observe that (*b*) includes new playback controls).

5.1.2.2. Backtracking. The exploration algorithm backtracks to a previously visited configuration when the explored configuration has been visited previously or when it has no successors (Listing 1, lines 22 and 36). Listing 3 shows a simplified Java code of the backtracking algorithm. The backtracking process consists of finding a configuration in `path` (from last to first) that has at least one unexplored successor (lines 4–13).

If `path` is not empty after this process, then the last configuration in `path` has a successor that has not yet been explored and corresponds to the next configuration that will be extracted from `unvisited` (Listing 1). In this case, the app controller closes the app and performs the list of user events (on their corresponding elements) included in `path`. When the app controller ends, the app will be running on the device and ready to accept the user event of the next unexplored configuration. Otherwise, if the path is empty, the app controller closes the app and throws an exception to indicate the end of the exploration.

```
 1  void backtracking(Configuration s, List<Configuration> path) throws
        Exception {
 2          Configuration s1;
 3          s.explored = true;
 4          while(!path.isEmpty()){
 5              s1 = path.getLast();
 6              if(allSuccesorsExplored(s1)){
 7                  s1.explored = true;
 8                  path.removeLast();
 9                  s1 = path.getLast();
10              }else{
11                  break;
12              }
13          }
14          AppController.closeApp();
15          if(!path.isEmpty()){
16              AppController.sendUserEvent(path)
17          }
18          else{
19              throw new Exception("No more branches to explore");
20          }
21      }
22  }
```

Listing 3. Backtracking algorithm

5.1.3. Model parser. The model parser produces the interaction model that will be used to generate the app user flows. The model parser interacts with the exploration algorithm when new states and/or transitions are added (Listing 1, lines 19, 25 and 26). When the exploration algorithm ends, the model parser dumps the interaction model into a file, using an XML-based internal representation of the modelling language presented in Section 2.2. The model parser is based on Velocity[‡‡], a Java template engine that generates the XML representation of the interaction model from a template.

It is worth mentioning some limitations of the current implementation. Although the modelling language can capture user and system events and delays between events, the automatically generated models do not include temporal delays in transitions or interactions with other apps. However, app user flow requirements can constrain the time elapsed after firing a particular event, which in practice is equivalent to including the delay in the model. Figure 3 shows the graphical representation of the Universal Music Player model.

[‡‡]http://velocity.apache.org/.

Table I. Model extraction – Configuration.

App	Max depth	List items	Events	el. excluded
UAMP	10	1	click	0
iDo Calc.	5	1	click	0
Kolab Notes	8	2	click	2
Topeka	15	1	click	0
Word Press	8	1	click	7

5.1.4. Evaluation of the model extraction. This section presents an evaluation of the model extraction approach using the depth-first search exploration strategy. The evaluation has been performed using a Samsung Galaxy S4 smartphone (non-instrumented) with Android 5.0.1 and a Windows 8 machine with an Intel Core i-7 3.600 GHz. The approach has been applied to five different apps, all of which were obtained from the Google Play Store or the Android SDK. The configurable parameters are the maximum depth explored (number of consecutive user events), the number of list items explored and the GUI elements excluded from the exploration. In addition, specific text can be associated with text input elements (e.g. a login form). Table I shows the configuration of these parameters for each app.

Universal Music Player (UAMP)[§§] is a music player that presents to the user a list of songs that can be streamed. From the point of view of the interaction model, playing any of the songs has the same effect. Thus, the model extraction is set to explore only the first item of each list. Note that UIAutomator has the limitation that it cannot obtain the DOM when the UI is changing dynamically, for instance, if a video is playing. Thus, the application controller (Section 5.1.1) has to pause the media playing before obtaining the DOM.

iDo Calculator[¶¶] is a calculator with simple and scientific modes. Text input is performed by clicking independent buttons for each digit and arithmetic operation. The maximum depth has been set to 5 to obtain a model that considers arithmetic operations with 1 or two digits.

Kolab Notes[‖‖] is an app for taking notes that can be local to a device, or shared by multiple devices using an account. The model considers only the local mode, and the controls to share the notes were excluded from the exploration. In addition, a colour picker component was excluded because it is currently difficult to handle this component in the model. With respect to the text input, sample text was provided for the note title and content.

Topeka[***] is an app for performing quizzes. Generated quiz questions are random; some are answered by choosing from four possible answers and others by writing text. To increase the probability of answering both types of questions during the exploration, the maximum depth was set to a high value.

WordPress[†††] is an app for visualizing and managing WordPress sites. This app has an initial login form, and thus, the form associates specific text with a valid user name and password. The app provides links to recover the user name and password, create a new account or read the Terms of Service, and these controls were excluded from the exploration. In addition, the app suggests a list of sites to visit, which changes dynamically and includes different clickable elements. This situation is similar to the dynamic questions in Topeka. However, to produce a first version of the model, the number of sites explored was limited by the list items explored and the maximum depth.

Table II shows some results. Because the max depth of the algorithm was limited, the resulting models may be a partial or incomplete representation of the app. On the one hand, for the UAMP app, increasing the depth of the algorithm produces the same interaction model, which implies a good representation of the app. On the other hand, the models of Topeka and WordPress, whose behaviour have a random component, are partial. In addition, different models can be extracted in

[§§]Available at https://github.com/googlesamples/android-UniversalMusicPlayer.

[¶¶]Available at https://play.google.com/store/apps/details?id=com.ibox.calculators.

[‖‖]Available at https://play.google.com/store/apps/details?id=org.kore.kolabnotes.android.

[***]Available at https://github.com/googlesamples/android-topeka.

[†††]Available at https://play.google.com/store/apps/details?id=org.wordpress.android.

Table II. Model extraction – Results.

App	Activities	St. Machines	States	Transitions	Time (min)	App launch
iDo Calc	2	2	9	93	49	23
UAMP	3	4	13	41	14	22
Kolab Notes	2	2	16	69	32	45
Topeka	3	3	16	51	33	38
Word Press	14	16	31	77	54	39

different executions of the model extraction algorithm. However, all these models present common interaction patterns. Thus, it is recommended to manually tune the models to allow the common patterns. Finally, the iDo Calculator model considers a click on each button as a different user event. Although all these events leave the app in the same state, the number of transitions is too high to produce a set of app user flows. In this situation, it is recommended to modify the model to abstract the buttons, for example, by defining two types of buttons, numeric and arithmetic, and thereby reducing the number of transitions.

5.2. *App user flow generation*

This section describes how to automatically produce app user flows, that is, sequences of user actions, using a model-based approach. The process starts with a model of the user interaction with the app, provided by the developer or automatically extracted from the compiled app as explained in Section 5.1. In both cases, the model is specified using the language described in Section 2.2.

The interaction model is explored exhaustively to generate all possible sequences of user actions. For this step, the power of the SPIN model checker [8] is used. SPIN can be used to analyse the correctness of concurrent software modelled using the PROMELA specification language. The focus of the tool is on the design and validation of computer protocols, although it has also been applied in other areas. SPIN checks the occurrence of a property over all possible executions of a system specification and provides counterexamples when violations are found.

```
#define cBackstack  devices[device].backstack
#define cState      cBackstack.states[cBackstack.index]
proctype device_91e1b2db(int device) {
    do
       // Entering state machine State_MusicPlayerActivity_0
    :: starting || cState == State_MusicPlayerActivity_0_init ->
        pushToBackstack(device, State_MusicPlayerActivity_0_init);
        transition(device, VIEW_MusicPlayerActivity, 0, EVENT_none);
        cState = State_MusicPlayerActivity_0_S0;
        // ...
    :: !starting && cState == State_MusicPlayerActivity_0_S6 ->
        transition(device, VIEW_MusicPlayerActivity, 22, EVENT_fullScreen);
        cState = State_FullScreenPlayerActivity_0_init;
        // Entering state machine State_FullScreenPlayerActivity_0
    :: !starting && cState == State_FullScreenPlayerActivity_0_init ->
        pushToBackstack(device, State_FullScreenPlayerActivity_0_init);
        transition(device, VIEW_FullScreenPlayerActivity, 35, EVENT_none);
        cState = State_FullScreenPlayerActivity_0_S12;
        //...
    :: !starting && currentState == State_MusicPlayerActivity_0_end ->
        popFromBackstack(device);
        continueTransition_91e1b2db(device);
    // State machine: FullScreenPlayerActivity_0
    // ...
    od;
    end_device_91e1b2db: devices[device].finished = true;
}
```

Listing 4. Extract of Universal Music Player PROMELA specification

In this case, SPIN performs an exhaustive exploration of the interaction model by translating the XML representation into a PROMELA specification. Each device is represented by a PROMELA process that models all of the state machines contained in that device. While the interaction 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 performed next.

To implement connection states, each device keeps a *backstack*, that is, a record of the transitions made in the past that lead to connection states. When a state machine finishes, the top of the backstack is checked. If it contains a connection state, then that information can be used to restore the next state in a previous state machine. If it is empty, then the execution of the model terminates.

This specification is explored depth-first by SPIN and each transition taken is recorded. When a valid end-state is reached, the sequence of transitions during the exploration contains the user actions, that is, an app user flow. If more than one transition can be chosen 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 app user flows will be generated.

Listing 4 presents a simplified excerpt of the PROMELA specification generated for the Universal Music Player app, showing some transitions of the MusicPlayerActivity_0 state machine. The process starting at line 3 models all the state machines contained on a device. The `transition` function (e.g. line 8) records each transition and updates the top of the backstack. To enter a new state machine from a connection state, its initial state is set first (line 13), and a new element is then pushed to the top of the backstack (line 16). To leave a state machine, the top of the backstack is popped (line 21), and the new top is checked to restore the state of the previous state machine (line 22).

```xml
<?xml version="1.0" encoding="utf-8" ?>
<appUserFlowRequirement
    xmlns="http://www.morse.uma.es/appuserflowrequirement"
    name="UAMP_never_2">
    <description> </description>
    <application codeName="uamp"/>
    <device serialNumber="91e1b2db"/>
    <invariants>
        <loop min="0" max="2"/>
        <state name="S1" visit="false"/>
        <event name="back" max="2" time="10"/>
        <view name="FullScreenPlayerActivity" visit="false"/>
        <view name="PlaceholderActivity" visit="false"/>
    </invariants>
    <sequence>
        <constraint type="simple">
             <state name="S6" view="MusicPlayerActivity"
                 statemachine="MusicPlayerActivity_0"  visit="true"/>
        </constraint>
        <constraint type="simple">
             <state name="S7" view="MusicPlayerActivity"
                 statemachine="MusicPlayerActivity_0" visit="true"/>
        </constraint>
        <constraint type="simple">
            <state name="S8" view="MusicPlayerActivity"
                statemachine="MusicPlayerActivity_0" visit="true" />
        </constraint>
    </sequence>
</appUserFlowRequirement>
```

Listing 5. XML description of an app user flow requirement

5.2.1. Optimized generation of app user flows. The previous section explained the production of all possible app user flows from the interaction model. However, to evaluate an app against a set of specific KPIs in the TRIANGLE testing framework (Figure 1), app user flows must activate some features of the app under test. For instance, to test a video streaming app, the app user flows must start and stop the video playback. The term *app user flow requirements* refers to the constraints that the app user flows must satisfy to evaluate a KPI. In the example, the app user flow requirements are to click on the play button and later on the stop button.

In previous work [13], the authors present an approach to optimize the generation of app user flows. The idea is to verify the interaction model against a set of properties, so that counterexamples represent the app user flows that satisfy the requirements. For this purpose, the requirements are described as Büchi automata using a *never claim*, which is a special PROMELA process that monitors the execution of the interaction model. Two different ways of defining these never claims, called pruning and non-pruning, were evaluated. Since the pruning never claims obtained a better performance, in this paper, all app user flow requirements are described in this way.

Currently, the TRIANGLE testing framework supports app user flow requirements described in XML notation. Listing 5 shows an example of the XML notation. The requirements are classified as invariants or constraints. Invariants describe conditions over states, events, views (activities) and loops that must always be true. In the example, the invariant restricts the number of loops (line 7) and forces the app user flow to avoid state $S1$ (line 8) and the activities FullScreenPlayerActivity and PlaceholderActivity. In addition, only two back events (line 9) can occur in the app user flow, and after them, a 10-s delay is included until a new event is fired. Constraints are restrictions over states, events or activities that must eventually hold in a given order. In Listing 5, the app user flow must pass at some point through state $S6$, then through state $S7$ and finally through state $S8$. Constraints can be simple or complex. In the first case, the constraint affects only a state, event or activity. In the second case, the constraint is defined as a set of restrictions that must hold simultaneously.

```
// never.tmp - never claim obtained from appUserFlowRequirement

/* Definition of invariants */
#define MAX_EVENT_back 2
#define INV_MAX_EVENT_back (c_expr{numEvents(0, EV_back)<= MAX_EVENT_back})
#define INVARIANT INV_MAX_EV_back && INV_MAX_LOOP /* rest of invariants */

/* Definition of constraints */

#define St0_VISIT_MUSICPLAYERACTIVITY_0_ST_S6 (INDEX_B &&
    devices[0].backstack.states[devices[0].backstack.index]==
    State_MusicPlayerActivity_0_S6)
#define CONSTRAINT_St0 St0_VISIT_MUSICPLAYERACTIVITY_0_ST_S6
#define NEG_CONSTRAINT_St0 /*...*/

#define CONSTRAINT_St1 /*...*/
#define NEG_CONSTRAINT_St1 /*...*/

#define CONSTRAINT_St2 /*...*/
#define NEG_CONSTRAINT_St2 /*...*/

never UAMP_never_2{
St0 : do
      :: INVARIANT && CONSTRAINT_St0 -> goto St1;
      :: INVARIANT && NEG_CONSTRAINT_St0
      od;
St1 : do
      :: INVARIANT && CONSTRAINT_St1 -> goto St2;
      :: INVARIANT && NEG_CONSTRAINT_St1
      od;
St2 : do
      :: INVARIANT && CONSTRAINT_St2 -> assert(false);
      :: INVARIANT && NEG_CONSTRAINT_St2
      od;
}
```

Listing 6. Never claim

The interaction model and the app user flow requirements are given as XML files and have to be compliant with their respective XML schemas. If they are not compliant, they cannot be translated to PROMELA and the automatic generation of app user flows cannot continue. Listing 6 shows the pruning never claim equivalent to the requirements of Listing 5. The body of the never claim (lines 20–33) is a sequence of `do` blocks, each one with two guarded branches. The first branch is guarded by the invariant and the constraint that must hold at some point, and if it is satisfied, the execution jumps to the next block. The second branch is guarded by the invariant and the pruning condition, which is essentially the negation of the constraint. If the guard is satisfied, the never claim remains in this `do` block. The final `do` block is slightly different: the first branch executes an `assert(false)` instead of a `goto` statement. Figure 6 presents a simplified definition of the invariants and the constraints (lines 4–18). Note that these definitions include references to variables and constants from the PROMELA model (Figure 5) and calls to auxiliary functions written in C. For example, `numEvents` returns how many times a specific event has been fired in the current app user flow.

The verification is carried out by the SPIN model checker. If the requirement contains references to non-existing elements in the interaction model, such as misspelled event or state names, SPIN reports the syntactical errors. The SPIN model checker is configured to detect assertion violations and does not stop when an error is found. In this way, it produces a counterexample each time the `assert` statement is executed, and each counterexample is an app user flow that satisfies the requirements.

5.3. Execution of app user flows and test cases

This section describes how the app user flows are automatically executed. To execute the app user flows, Quamotion WebDriver was used again [17]. Because it implements the standard WebDriver protocol, a number of clients can interface with it. In TRIANGLE, the Keysight Testing Automation Platform (TAP) [26] automates the whole process. TAP is a flexible platform that orchestrates the interaction of the testing framework with a number of instruments that interact around the device under test (DUT). A new TAP plugin has been developed to interact with the Quamotion WebDriver, which executes an app user flow written as a JSON script. The TAP plugin translates each of the user actions into calls to the WebDriver API. Before executing an action, the plugin checks and waits for the action to be executable, for example, that the referenced `xPath` is indeed available in the current contents of the screen. The TRIANGLE testing framework monitors the execution of the app user flows, measures different parameters (average current, transmitted and received data, user events, etc.) and stores this information in a database.

5.4. Verification of extra-functional properties

The final step of this approach is the analysis of the test case execution. This step involves the analysis of execution traces to verify the given EFPs.

The TRIANGLE testing framework provides information about the device on which the app runs (such as the average power, transmitted/received data and transmission/reception rate) and the network conditions (cells, eNodeBs, etc.). In addition, the developer can instrument the app under test to include in the traces when certain events or pieces of code are executed, which is needed during the runtime verification process of EFPs. We call these new traces *enriched traces*.

Each EFP is transformed into a monitor whose implementation is described in the succeeding text. The EFP monitor analyses the trace on the fly while the test case is running, and it returns a verdict as soon as one is available. Currently, the TRIANGLE testing framework provides the enriched trace after the test case ends and, as a consequence, the monitor analyses it offline.

The EFP monitor carries out two interrelated tasks. On the one hand, it has to read the states of the enriched trace and, on the other, it has to simultaneously evaluate these states against the EFP. These two tasks are implemented using a PROMELA specification with embedded C code such as the one shown in Listing 7. Thus, the PROMELA monitor contains (i) code to translate on the fly the enriched trace into a sequence of states [27] that constitutes the *model* analysed by SPIN (for example, the code of Listing 7) and (ii) code that implements the evaluation of the EFP, which is implemented as a *never claim* PROMELA process from the LTL formula, as described in Section 3.3. The *never claim* process corresponds to the Büchi automaton of the LTL formula in SPIN notation. The tool

SPIN automatically transforms (the negation of) the LTL specifications into *never claim* processes and executes them with the system model in a synchronized manner. This means that SPIN always fires a transition in the model followed a transition in the *never claim* and that naturally acts as a observer of the system execution searching for erroneous behaviours.

In the following, Listing 7, which shows a simplified fragment of the PROMELA specification for the next formula, is used to detail how the SPIN model checker evaluates the formula on an enriched trace.

$$\forall\, [[diff\_energy < P]]_{[testStep=START,testStep=END]}$$

```
 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 7. Fragment of a PROMELA EFP monitor

The PROMELA specification contains global variables for each of the relevant variables in the trace. The core of this specification is a loop (lines 20 to 35) that reads and reconstructs the trace. 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 correctly restore previously visited states. From SPIN's point of view, each new iteration leads to a new state to be explored. As commented earlier, to analyse the EFPs, the corresponding LTL formula (Equations 1, 2 and 3 of Section 3.3) is negated and translated into a *never claim* process.

Additional global variables and helper functions are used to address continuous variables and interval properties. The current value of a given continuous variable c is stored in a floating point variable `c`. These variables, like any other, are updated on each iteration of the core loop. The example has one continuous variable, called `energy` (line 3). To evaluate an interval formula Φ in an interval $[t_1, t_2]$, the initial and final values of the continuous variable c, that is, $c(t_1)$ and $c(t_2)$, must be available. While the former is already stored in the global variable `c`, a new global variable `c_t1` is needed for the latter. These variables are updated automatically by the so-called `update` functions, that is, C functions that compute the values of variables that are obtained from the enriched trace.

Line 13 shows the update function for the `energy_t1` variable (line 4). This function updates the values of `energy_t1` only at the start of a new interval. Another variable `_interval`, which is automatically updated by another function (line 7), is introduced to detect the intervals. 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. EVALUATION

In previous work [12], the authors applied the proposed model-based methodology to analyse the network traffic produced by the Spotify app for Android devices. The implementation of this approach[‡‡‡] was not integrated into the TRIANGLE testing framework, and some of its parts, such as automatic model extraction or the optimization of app user flows, were not yet designed and implemented. This section presents the evaluation of the approach using a real app as case study.

6.1. *Description of the case study*

The Universal Music Player is the application under test that will be used as case study. It is music streaming app included in the Android Studio that includes a fixed play list. The main GUI components are presented in Section 2.1 and Figure 2. When the user selects a song to play, the app send a request to the music servers and start the download of the song and the playback. Clearly, the network (network load, coverage, user mobility, etc.) can influence the quality of service, which is reflected for example in the traffic transferred between the music servers and the app. Thus, the case study objective is to evaluate if the UAMP satisfies the following EFP: 'during the playback of the first song of the playlist, the data received by the app are under a threshold'. If the EFP is satisfied, then there is a low number of packet retransmissions during the playback.

6.2. *Experiment set-up*

This section summarizes the configuration used in the evaluation of each of the three phases of the approach. The first phase consists of extracting of the interaction model from the app binaries, which is independent of the TRIANGLE testbed and can be carried out manually or automatically. The automatic model extraction has been performed using a Samsung Galaxy S4 (non-instrumented) with Android 5.0.1 and a Windows 8 machine with an Intel Core i-7 3.600 GHz. In addition, the exploration algorithm was configured with a maximum depth of 10 *click* events, and in case of list, only the first item of lists is explored.

The second phase is the automatic generation of app user flows that activate the features necessary to analyse the EFP. The specification of the app user flow requirement has to be consistent with the interaction model obtained in the first phase. To facilitate this task, the (automatically extracted) interaction model was modified to include more intuitive and unified event names, for example, clicking the play/pause button or the back button takes place in multiple transitions. Figure 3 shows the UAMP interaction model with the modified event names.

The app user flow requirement must at least start playing a song during a time interval and then pause, which can be expressed in different ways. Three different app user flow requirements have been defined in order to ensure the playback of video during the test case execution as well as other extra requirements to reduce the number of app user flows generated. In addition, the maximum app user flow length is configured to 10, which means that the app user flows will have, at most, 10 events.

[‡‡‡]http://www.morse.uma.es/tools/mve.

- Req. 1: The app user flow must eventually reach state $S6$ with the event *selectSong* and then eventually reach $S6$ with the event *play/pause*. In addition, each previous event can happen only once.
- Req. 2: The app user flow must eventually reach state $S6$ with the event *selectSong* and then eventually reach $S6$ with event *play/pause*. In addition, loops are not allowed, and FullScreenPlayerActivity and PlaceHolderActivity are never entered.
- Req. 3: The app user flow must reach state $S6$ with the event *selectSong*, then eventually fire the event *skipNext* and finally the event *play/pause*. In addition, loops are not allowed, and states $S1$ and $S9$ are not visited.

The last phase is the execution of the test case and the analysis of the EFP in different network scenarios. The app user flow (obtained in the previous phase) satisfies Requirement 3. The app user flow starts playing a song, then skips to the next song and finally pauses the playback.

Figure 7 shows the TRIANGLE platform, which includes a mobile network testbed that emulates complex radio and network conditions. The EFP included in the case study is especially relevant in network scenarios with a high density of users and a heavily loaded network. This issue has been considered to select the following network scenarios for the case study:

1. The urban pedestrian scenario emulates a user walking down an urban street at 1 to 3 km/h. This scenario is considered as normal conditions.
2. The suburban festival scenario reproduces an outdoor event with many attendees, and thus, some small cells are deployed to increase capacity.
3. The Internet café busy hours emulates an Internet café during busy hours in a dense urban area, and thus, there is a dense number of users in a reduced area.

The TRIANGLE testing framework executes automatically generated app user flows to start and pause the playback in three different network scenarios, and the resulting execution traces are used to verify the EFP. Due to time and space limitations, each test case is executed twice.

The last two phases, the automatic generation of app user flows followed by the test case execution and the analysis of EFPs, are carried out in the TRIANGLE testbed, which emulates realistic network scenarios that are wholly reproducible. In addition, the TRIANGLE testbed supports the traceability and repeatability of the test. In this case, a rooted Samsung Galaxy S7 (instrumented) with Android 6.0.1 has been connected to the testbed to execute the tests.

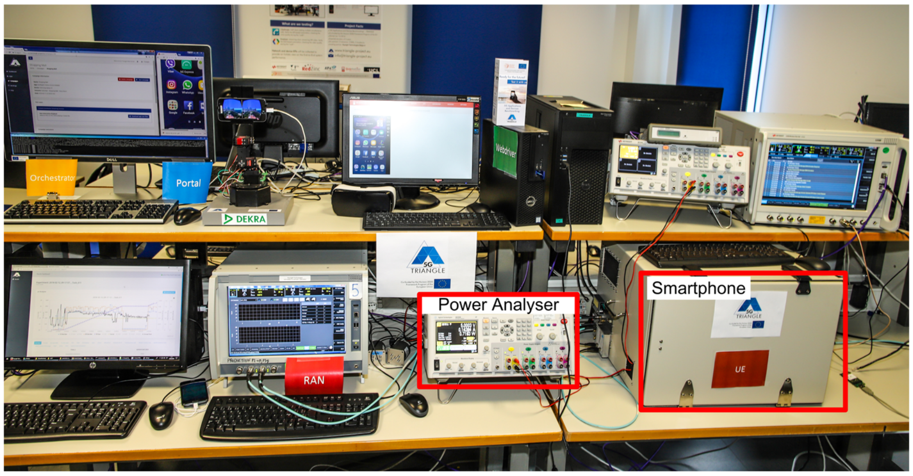

Figure 7. Smartphone connected to the TRIANGLE testing framework.

Table III. App user flow generation statistics.

	No Req.	Req. 1	Req. 2	Req. 3
State vector	144	148	148	148
Depth	151	183	175	176
States stored	27438	17624	12858	11542
Transitions	27438	22287	16097	14285
Atomic steps	57357	26283	20519	18875
Total mem.(MB)	6.543	8.168	7.484	7.289
Time elap. (s)	0.517	0.428	0.468	0.437
App user flows	-/7906	**260**/4663	**232**/3239	**268**/2743

6.3. *Analysis of the results*

Section 5.1.4 included the evaluation of the model extraction technique using five different apps as case studies, including the UAMP. This section presents the results obtained in the second and third phases of the complete approach for the UAMP case study.

6.3.1. App user flow generation. Table III shows the statistics returned by SPIN and the number of app user flows generated. The second column shows the statistics of exhaustive exploration of the model without requirements, and the other three show the statistics for each requirement. The last row includes two numbers. The first is the number of app user flows that satisfy the requirement, and the second is the number of traces explored. In the previous work [12], the number of app user flows generated was the number of app user flows explored without using requirements, which did not ensure the activation of the app feature required to analyse the EFP. The definition of app user flow requirements produces only the relevant app user flows and generally takes less time.

6.3.2. Test case execution and trace analysis. The execution of a test case entails the configuration of the TRIANGLE testing framework, the execution of the app user flow, the post-processing of the result and the verification of the EFPs. In this example, the time elapsed is approximately 10 min for each test case. Due to time and space limitations, each test case was executed twice.

The TRIANGLE testbed returns six different execution traces. Figure 8 shows the data received over time for each of the six traces, from the beginning of the test cases until the pause event is captured. The slopes of the graphs show that in heavily loaded network scenarios (festival and Internet café), the data are received slowly, with a smoother slope, than in the pedestrian scenario. These graphs are analysed against the EFP expressed in the following formula:

$$\forall\,[[diff\_rxData < 8000(kb)]]_{[StartPlaying,PausePlaying]}$$

This formula states that during the playback (events StartPlaying and PausePlaying), the traffic received is always under a threshold of 8000 KB. This threshold is the baseline obtained in ideal conditions. Figures 9, 10 and 11 show the received data and the StartPlaying and PausePlaying events of three different execution traces. Clearly, the EFP is satisfied in the pedestrian trace. However, in the suburban and the Internet café scenarios, the threshold is exceeded. This behaviour can be caused by the retransmission of packets due to network congestion. With this information, an app developer or service provider can decide to implement an adaptive streaming algorithm that modifies the audio quality based on network congestion to reduce the traffic load. However, it is not the objective of the authors to determine whether UAMP implementation should be improved.

6.4. *Threats to validity*

The authors have detected two main threats to validity during the evaluation. The first one is related to the automatic extraction of the interaction model. As commented in Section 5.1.4, the model extracted can be a complete or partial representation of the user interactions with the app. This problem arises due to the non-determinism of some apps or the use of very dynamic content. In these cases, the automatic model extraction approach can produce different interactions models, and

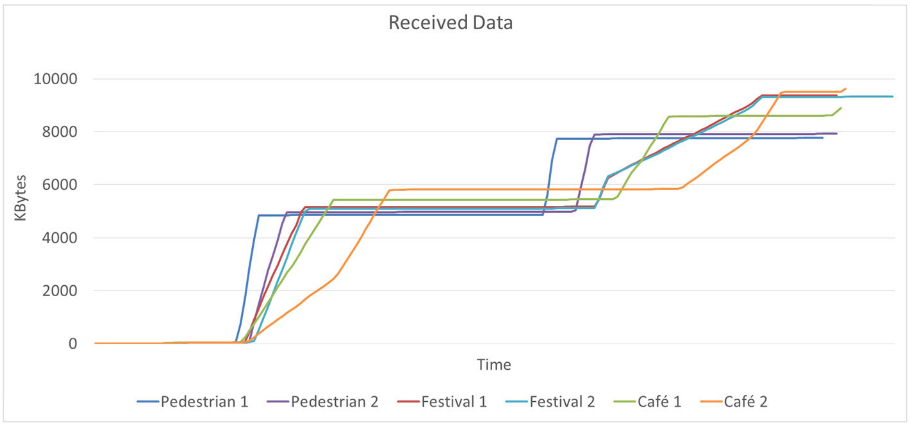

Figure 8. Received data – all execution traces.

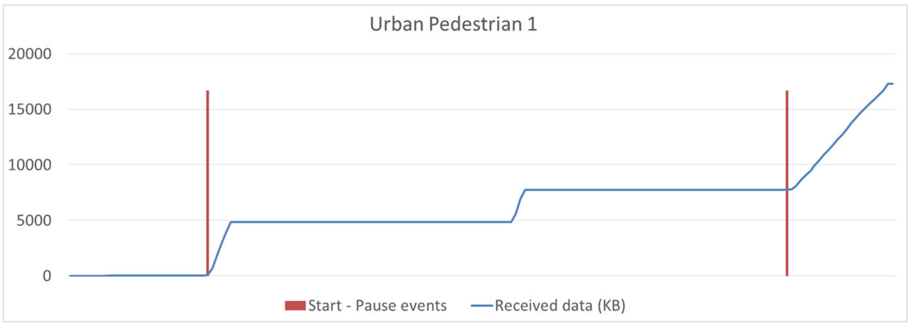

Figure 9. Received data – urban pedestrian, first execution.

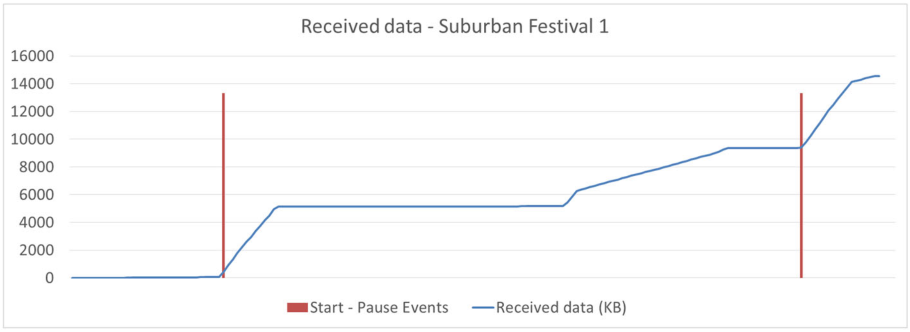

Figure 10. Received data – suburban festival, first execution.

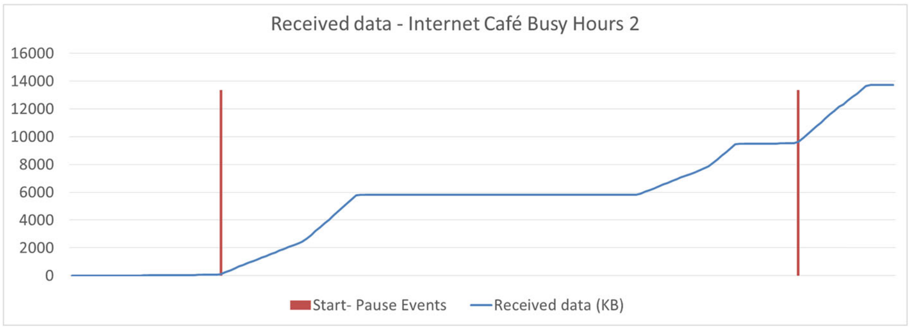

Figure 11. Received data – Internet café busy hours, second execution.

some fine tuning of the model is required. However, the authors have observed that it is possible to infer some common interaction patterns from the different models that can help the app developer to manually tune or extract the model.

The second threat is the test coverage and the accuracy of the statistical results. Currently, the TRIANGLE testbed can automatically run longer test campaigns with more network scenarios and execute different app user flows selected from the pool of app user flows obtained in the second phase. However, due to space and time limitations, the evaluation only considers the execution of one app user flow two times per network scenario. More test cases must be executed to improve the coverage of the results. It is worth mentioning that further results of ongoing work to apply the proposed technique to commercial apps will be reported at TRIANGLE website.

7. COMPARISON WITH RELATED WORK

There are some existing proposals that apply model-based testing to Android applications. Some of them assume that the testing process starts without a precise model of the expected behaviour of the applications and focus on techniques to obtain such a model. The *MobiGUITAR* framework [28] automatically constructs a state machine of one application by executing events in the running application and recording a tree with fireable events for each new state. The authors use a 'breadth-first' traversal of the app GUI for open source applications. They do not consider any knowledge about how to use the application but instead conduct an exhaustive execution. Therefore, they need criteria to make some states equivalent to prevent state explosion.

The *Swift-Hand* technique proposed by Choi et al. [29] employs machine learning to construct an approximate model of the application during the testing process. Their aim is to cover as much behaviour as possible, forcing the execution to enter unexplored parts of the state space. Other approaches also use a formal specification of the application to start the test generation. Jing et al. [30] describe how to follow a property-driven method to build models in Alloy, a formal language based on first-order logic. In their proposal, the role of the model checker in this approach is performed by the Alloy analyser, which generates positive (expected) and negative (undesired) test cases.

In contrast to *MobiGUITAR*, this approach separates test generation from testing, and the states in the high-level state machines are limited and differentiated by design. Thus, these models are more compact. For example, compared with *MobiGUITAR*, it is unnecessary to conduct extra work to remove unrealistic test cases. In addition, this approach allows the generation of test cases for several applications that interact using Android intents, while the complexity of the runtime-based modelling process for *MobiGUITAR* and *Swift-Hand* makes them more suitable for single applications. Similar to this approach, Jing et al. [30] use XML-based transformations to translate the test cases to an executable form to activate the applications being tested. Apart from the inner technologies (model checking vs constraint solver), the main difference between the proposals is how the refined executable model is obtained. The Alloy specification presented by Jing et al. [30] is constructed manually, while the PROMELA specification in this work is generated automatically from the high-level design of the user-view state machines. Both should be applied to the same case study to obtain a quantitative comparison of the human and computational effort required in these two approaches.

Many commercial and academic tools have been developed to monitor and analyse certain types of EFP in mobile phones. One group of tools focuses on traffic analysis. AntMonitor [3] is a powerful tool that monitors all connected apps in the Android device to produce statistics, such as how each app contributes to the total network traffic, but does not monitor specific events or user interactions systematically. NetworkProfiler [4] is a tool designed to help cellular operators identify the traffic in their networks when transported over HTTP/HTTPS. NetworkProfiler uses device emulators and machine learning techniques to create an app fingerprint by collecting information on the hosts to which the app connects. Because the manual exploration of apps being tested will not cover all behaviours, NetworkProfiler randomly generates user actions to interact with the apps. This random strategy is also supported by other tools such as Monkey [19]. Other tools, such as Monkeyrunner [20], Robotium [21] and Troyd [31], employ scripts to make the apps work with a predefined sequence of user interactions. However, they do not provide an automated methodology to generate such scripts. ProfileDroid [5] is designed to systematically profile apps to discover inconsistencies or surprising behaviours. It is based on a multilayer analysis of the apps, considering the static analysis of byte code, user interactions, calls to the operating system and network traffic. This tool requires a real user to interact with the phone, but the sequence of interactions can be recorded and replayed later in a different scenario. The approach of Automatic Android App Explorer (A3E) [1] involves the novel use of static dataflow analysis on the app bytecode to construct a high-level control flow graph that captures legal transitions among activities (i.e. app screens). Depth-first exploration is then used to reach 64 per cent activity coverage and 36 per cent method coverage in some typical Android apps. This flow graph plays the same role as the models in this study that represent user interactions, and the depth-first search is similar to the approach used in this study for exhaustive test generation. ProfileDroid and A3E are the closest proposals to this work and could in fact be adopted in this methodology.

Compared with ProfileDroid and A3E, this methodology has the following main novel points: (i) the controlled coverage of user interactions for one or several apps due to the model-based approach for test generation; (ii) the ability to automatically find execution traces that violate the expected behaviour of the app in terms of their effect over the Internet; (iii) the inclusion of time in the models making possible to test realistic situations in which the time between user interactions is relevant; (iv) the ability to combine models of several apps running in parallel; and (v) a technique to automatically compare the actual behaviour of each execution of the apps with the expected behaviour. ProfileDroid could be used to generate the reference patterns employed to analyse the actual behaviour of the apps. A3E could be used to facilitate the construction of the interaction models.

There are many references to existing research on one of the EFPs highlighted in this paper, namely estimating power consumption. Phatak et al. [32] provided one of the first classifications of energy bugs for hardware and software and proposed a roadmap towards developing a systematic diagnostic framework for treating these energy bugs. Later, Phatak and other authors presented the **eprof** tool [33], a fine-grained energy profiler used to gain insight into the energy usage of smartphones. Simultaneously, Yepang Liu [2] studied energy bugs (e.g. their types and manifestation) and identified common patterns. They implemented a static code analyser, **PerfChecker**, to detect and identify bug patterns. The **E-loupe** project [34] explores an alternative that mitigates the ill effects of an energy-hungry application. The framework consists of monitoring data in the mobile phone, which are then processed in the cloud to detect the risk of energy drain and to produce information to isolate the dangerous applications. Memory leaks can also be a cause of energy consumption. Xia et al. [35] design a light memory leaks detector that focuses on activity leak and a priority adjustment module to prioritize the killing of leaking apps. In a different approach, a framework is built by Zhang et al. [36] to detect energy leaks using dynamic taint analysis (a form of information flow analysis). Finally, the energy consumed in different mobile platforms is studied in different works [37–39]. The first approach compares energy bugs, the second compares energy efficiency and the last developed a power estimation method based on battery traces.

The model-based framework in this study, also required energy consumption monitoring, similar to most of the aforementioned proposals. However, this output is not used directly, but instead as an input for a more sophisticated analysis. Interval logic is used to represent the energy properties to drive the identification of bad behaviours by the applications running the smartphone. As a result, this verification technique is able to detect leaking apps in a very precise way: it can provide the exact execution sequence of one or several apps that are causing the system to lose more energy than expected.

Thompson et al. present a methodology and tool [40], called SPOT, to model the architecture of an application and emulate its energy consumption during the design phase. The application is modelled using configurable abstractions of elements that typically contribute to energy consumption, such as GPS and network connections, attached to activities or background services. The SPOT tool running on the Android device uses this model to emulate each of the components. The energy consumption of an emulation is logged using an Android API and provided as feedback to the programmer.

While SPOT also uses a model-driven approach, the tool presented here is oriented towards modelling user behaviours and using the real application code instead of a simplified architecture and emulated components. Together with the TRIANGLE measuring and monitoring modules, this approach provides a more accurate analysis of an application's real power consumption profile. In addition, the energy measurements can be correlated with other real runtime sources, such as network traffic or debug information, to analyse other functional properties or EFP.

Finally, the use of model checking with EFPs has also been applied to estimate energy consumption in wireless sensor networks. Schmitt and Werner [41] use a UPPAAL model of the network that includes aspects such as time, bandwidth and energy. Then, they use model checking with different scenarios to predict the influence of a given set of parameters on the energy consumption. In particular, they focus on sensors and routers. Nakajima [42] recently introduced two new formalisms to address with this problem, the *power consumption automaton* to represent the system under analysis and a version of LTL with freeze quantifiers to represent expected energy consumption. The presence of time and energy in the models and formulae make this model checking problem undecidable, and the author proposes several practical subsets to run verification. Compared with these proposals, the approach proposed in this study upholds the idea of one specific logic to represent energy properties and the model checking mechanics to check this logic. In addition, instead of exploring a model of a system, this study addresses execution traces extracted from real devices with precise energy consumption measures for each sampling period.

8. CONCLUSION AND FUTURE WORK

This paper presents the foundation for a complete platform to verify the EFP of Android mobile applications. The method is based on two formal languages to specify (i) user–application interactions as the input for the automatic generation of test cases and (ii) the expected EFPs that the executions of the app should satisfy. Both test generation and verification are performed with model checking. In addition, a tool has been implemented to assist the developer in the modelling phase. This tool automatically extracts the model of an app, which can be edited later to add/remove behaviours that the tool is not able to capture due to the inherent complexity of apps. This paper also presents an implementation, which is integrated in the TRIANGLE testing framework, and an evaluation of the complete framework with a realistic application.

In future work, the implementation should be adapted to the iOS operating system. Due to Quamotion WebDriver's compatibility with iOS devices, most of the components could be reused with minor or no changes, like the language for EFPs and the mechanisms to generate test cases, and the automation of user events. However, it will be necessary to determine whether the management of activities/screens in iOS can be captured with the proposed modelling language. To facilitate the definition of app user flow requirements, the authors plan to implement an editor supporting the modification of the interaction model as well as the generation of associated app user flow requirements. To this end, the definition of templates or requirements meta-models would be required. In addition, it would be desirable to define a complete interval logic that allows the specification of richer properties. Moreover, the inclusion of runtime verification of EFPs in the TRIANGLE testing framework will be examined. The objective is to stop the test case execution if the EFP is not satisfied and to start the analysis of the next test case as soon as possible.

ACKNOWLEDGEMENTS

Work is partially supported by the Spanish Ministry of Economy and Competitiveness project TIN2015-67083-R. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 688712 (TRIANGLE project).

REFERENCES

1. Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM Sigplan International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, ACM: New York, NY, USA, 2013; 641–660.
2. Liu Y, Xu C, Cheung S-C. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM: New York, NY, USA, 2014; 1013–1024.
3. Shuba A, Le A, Gjoka M, Varmarken J, Langhoff S, Markopoulou A. AntMonitor: Network traffic monitoring and real-time prevention of privacy leaks in mobile devices. In *Proceedings of the 2015 Workshop on Wireless of the Students, by the Students, and for the Students*, S3 '15, ACM: New York, NY, USA, 2015; 25–27.
4. Dai S, Tongaonkar A, Wang X, Nucci A, Song D. NetworkProfiler: Towards automatic fingerprinting of android apps. In *INFOCOM, 2013 Proceedings IEEE*: Turin, Italy, 2013; 809–817.
5. Wei X, Gomez L, Neamtiu I, Faloutsos M. ProfileDroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, ACM: New York, NY, USA, 2012; 137–148.
6. Broy M, Jonsson B, Katoen J-P, Leucker M, Pretschner A. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag: Berlin, Heidelberg, 2005.
7. Havelund K. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer* 2015; **17**(2):143–170.
8. Holzmann GJ. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional: Boston, 2003.
9. Cattoni AF, Corrales-Madueño G, Dieudonne M, Merino P, Díaz-Zayas A, Salmerón A, Carlier F, Saint-Germain B, Morris D, Figueiredo R, Caffrey J, Polglase JB, Cardenas C, Roche N, Moore A. An end-to-end testing ecosystem for 5g. In *European Conference on Networks and Communications, EuCNC 2016*: Athens, Greece, 2016; 307–312.
10. Espada AR, Gallardo M-M, Salmerón A, Merino P. Using model checking to generate test cases for android applications. In *Proceedings Tenth Workshop on Model Based Testing* Pakulin N, Petrenko AK, Schlingloff B-H (eds), Electronic Proceedings in Theoretical Computer Science, vol. 180: Open Publishing Association: London, 2015; 7–21.
11. Espada AR, Gallardo MM, Salmerón A, Merino P. Runtime verification of expected energy consumption in smartphones. In *Proc. of the 22nd International Symposium on Model Checking Software (SPIN 2015)* Fischer B, Geldenhuys J (eds): Springer International Publishing: Stellenbosch, South Africa, 2015; 132–149.
12. Espada AR, Gallardo M-M, Salmerón A, Merino P. Performance analysis of spotify® for Android with model-based testing. *Mobile Information Systems* 2017; **2017**:1 –14.
13. Panizo L, Salmerón A, Gallardo MM, Merino P. Guided test case generation for mobile apps in the TRIANGLE project: Work in progress. In *Proc. of the 24th ACM Sigsoft International spin Symposium on Model Checking of Software*, SPIN 2017: ACM: New York, NY, USA, 2017; 192–195.
14. *UML 2.4.1 Superstructure Specification*, 2011. Technical report, Object Management Group (OMG).
15. Harel D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 1987; **8**(3): 231–274.
16. Chaochen Z, Hansen MR. *Duration Calculus - A Formal Approach to Real-Time Systems*, Monographs in Theoretical Computer Science. An EATCS Series. Springer: Berlin, Heidelberg, 2004.
17. Quamotion. *Quamotion WebDriver*. Accessed on 2018.03.21.
18. Tim B. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC Editor, 2017. https://rfc-editor.org/rfc/rfc8259.txt.
19. Android. *UI/Application Exerciser Monkey*. https://developer.android.com/studio/test/monkey.html [last accessed on 2018.03.21].
20. Android. *monkeyrunner*. https://developer.android.com/studio/test/monkeyrun-ner/index.html [last accessed on 2018.03.21].
21. Robotium Team. *Robotium: User Guide Android Studio*. https://robotium.myshopify.com/pages/user-guide-android-studio [last accessed on 2018.03.21].
22. Android Open Source Project. *UIAutomatorViewer*. http://developer.android.com/training/testing/ui-testing/index.html. Accessed on 2018.03.21.
23. Stewart S, Burns D. *WebDriver*: W3C Candidate Recommendation, W3C, 2017. https://www.w3.org/TR/webdriver/.
24. Amalfitano D, Amatucci N, Memon AM, Tramontana P, Fasolino AR. A general framework for comparing automatic testing techniques of Android mobile apps. *Journal of Systems and Software* 2017; **125**:322–343.
25. Clark J, DeRose S. XML Path Language (XPath) Version 1.0, W3C, 1999. [last accessed on 2018.03.21].
26. Keysight Technologies. *Test Automation Platform Developer´s System*. [last accessed on 2018.03.21].
27. Salmerón A, Merino P. Integrating model checking and simulation for protocol optimization. *Simulation* 2015; **91**(1):3–25.
28. Amalfitano D, Fasolino AR, Tramontana P, Ta B, Memon A. MobiGUITAR – A tool for automated model-based testing of mobile apps. *IEEE Software* 2014; **99**(PrePrints):1.
29. Choi W, Necula G, Sen K. Guided GUI testing of Android apps with minimal restart and approximate learning. *SIGPLAN Not.* 2013; **48**(10):623–640.

30. Jing Y, Ahn G-J, Hu H. Model-Based Conformance Testing for Android. In *Advances in Information and Computer Security - 7th International Workshop on Security, IWSEC 2012, Fukuoka, Japan, November 7-9 2012. Proceedings*: Berlin, Heidelberg, 2012; 1–18.
31. Jeon J, Foster JS. *Troyd: Integration Testing for Android*, 2012.
32. Pathak A, Hu YC, Zhang M. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X: ACM: New York, NY, USA, 2011; 5:1–5:6.
33. Pathak A, Jindal A, Hu YC, Midkiff SP. What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12: ACM: New York, NY, USA, 2012; 267–280.
34. Chandra R, Fatemieh O, Moinzadeh P, Thekkath CA, Xie Y. End-to-End Energy Management of Mobile Devices. *Technical Report MSR-TR-2013-69*, Microsoft, 2013.
35. Xia M, He W, Liu X, Liu J. Why application errors drain battery easily?: A study of memory leaks in smartphone apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13: ACM: New York, NY, USA, 2013; 2:1–2:5.
36. Zhang L, Gordon MS, Dick RP, Mao ZM, Dinda P, Yang L. ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12: ACM: New York, NY, USA, 2012; 363–372.
37. Metri G, Shi W, Brockmeyer M. Energy-Efficiency Comparison of Mobile Platforms and Applications: A Quantitative Approach. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15: ACM: New York, NY, USA, 2015; 39–44.
38. Wang C, Yan F, Guo Y, Chen X. Power estimation for mobile applications with profile-driven battery traces. In *2013 IEEE International Symposium on Low Power Electronics and Design (ISLPED)*: Beijing, China, 2013; 120-125.
39. Zhang J, Musa A, Le W. A comparison of energy bugs for smartphone platforms. In *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*: San Francisco, CA, USA, 2013; 25–30.
40. Thompson C, Schmidt D, Turner H, White J. 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*: Algarve, Portugal, 2011; 101–113.
41. Schmitt PH, Werner F. Model Checking for Energy Efficient Scheduling in Wireless Sensor Networks. *Technical Report Tech. report*, 2007.
42. Nakajima S. Model checking of energy consumption behavior. In *Complex Systems Design & Management Asia*, Cardin M-A, Krob D, Lui CP, Tan HY, Wood Ks (eds). Springer International Publishing: Cham, 2015; 3–14.