Paper Title: Guided test case generation for mobile apps in the TRIANGLE project: work in progress

Authors: Laura Panizo, Alberto Salmerón, María-del-Mar Gallardo, Pedro Merino[1]

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

Presented at: 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN 2017), 13-14 July 2017, Santa Barbara, CA, USA
Published in: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software
DOI (link to publication from Publisher): 10.1145/3092282.3092298
Publication date: 2017
Document Version Accepted author manuscript, peer reviewed version.

Guided Test Case Generation for Mobile Apps in the TRIANGLE Project: Work in Progress*

Laura Panizo, Alberto Salmerón, María-del-Mar Gallardo, and Pedro Merino
Universidad de Málaga, Andalucía Tech.
Dept. Lenguajes y Ciencias de la Computación
Málaga, España
[laurapanizo,salmeron,gallardo,pedro]@lcc.uma.es

ABSTRACT

The evolution of mobile networks and the increasing number of scenarios for mobile applications requires new approaches to ensure their quality and performance. The TRIANGLE project aims to develop an integrated testing framework that allows the evaluation of applications and devices in different network scenarios. This paper focuses on the generation of user interactions that will be part of the test cases for applications. We propose a method that combines model-based testing and guided search, based on the Key Performance Indicators to be measured, and we have evaluated our proposal with an example. Our ultimate goal is to integrate the guided generation of user flows into the TRIANGLE testing framework to automatically generate and execute test cases.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation*; *Software testing and debugging*; *Empirical software validation*;

KEYWORDS

Model-based testing, test case generation, Spin

ACM Reference format:
Laura Panizo, Alberto Salmerón, María-del-Mar Gallardo, and Pedro Merino. 2017. Guided Test Case Generation for Mobile Apps in the TRIANGLE Project: Work in Progress. In *Proceedings of SPIN'17, Santa Barbara, CA, USA, July 13-14, 2017,* 4 pages.
https://doi.org/10.1145/3092282.3092298

1 INTRODUCTION

The growing influence of mobile phone applications in our lives has revealed the importance of testing techniques that ensure their quality. Testing is not only important to detect behavioural or functional errors, but also to assess non-functional properties related to the applications performance or the user experience. Mobile apps have to deal with a highly changeable environment (changes in bandwidth, lost of connection, etc.) and with limited resources (battery, memory, etc.). For a developer, it is important to know how the application reacts in these adverse scenarios.

The TRIANGLE project[1] focuses on the development of a testing framework to ensure users' Quality of Experience (QoE) in 5G networks, since they will introduce very challenging network scenarios. Apps and devices will be benchmarked using a set of Key Performance Indicators (KPIs), organised around 5G use cases such as videostreaming.

In previous work [2–4], we presented an approach for automating the analysis of Android mobile apps, based on model-based testing and runtime verification. The former was used to generate a large set of test cases from an app model provided by the user, and then the latter analysed the executions of each one for certain properties of interest.

This approach has one fundamental problem: a reasonably complete model of an app will generate thousands, if not millions, of user interactions. This is unfeasible to execute on a real mobile device. Furthermore, if a developer wanted to test one particular part or feature of the app, the model had to be manually modified in order to include only the desired behaviours. The compositional nature of the app models was not enough to make this task easy, and the user could also miss significant behaviours that contributed to the feature being tested while modifying the model. Other approaches addressing test case generation define a coverage criteria that reduces the number of test cases [6], or use equivalent class partitioning techniques to generate a small and representative suite of test cases [1].

Our aim is to extend our previous work with similar techniques that allow the guided generation of user interactions that are useful for a given purpose, using only a *single* app model. In the case of the TRIANGLE project, KPIs will be used to guide this process. In this way, we explicitly separate the app model construction and the specific requirements to produce meaningful test cases. In consequence, the reduction of the generated user interactions happens in the generation process rather than in the modelling phase.

The paper is organised as follows. Section 2 introduces the TRIANGLE project. Section 3 presents our approach to reduce the number of test cases generated, and shows some preliminary results using a music player mobile app. Finally, Section 4 summarises the current and future work.

*The TRIANGLE project is funded by the European Union's Horizon 2020 research and innovation programme, grant agreement No 688712.

SPIN'17, July 13-14, 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5077-8/17/07.
https://doi.org/10.1145/3092282.3092298

[1] http://www.triangle-project.eu/

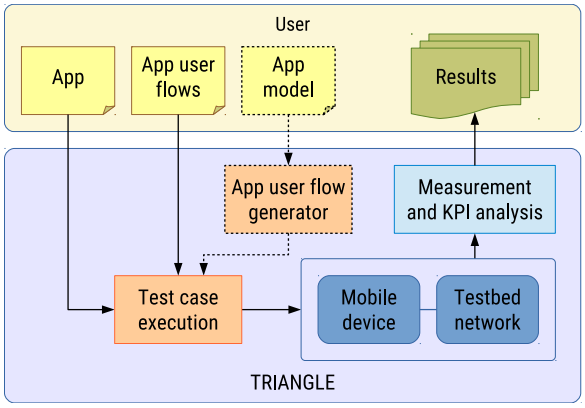


Figure 1: App user flows in the TRIANGLE testing framework overview

2 THE TRIANGLE PROJECT

The goal of the TRIANGLE project is to provide a framework where app developers and device makers can test and benchmark their applications and devices, to make sure they are ready for the challenges presented by the imminent 5G future. Developers will be able to evaluate their products and refine them in a controlled lab environment, and also opt for certification, comparing their performances against reference values for the use cases they support. Connected eHealth, smart cities, and media are among the uses cases that will greatly benefit by future 5G deployments.

TRIANGLE will take care of emulating several aspects of the mobile network, from the radio access with impairments to the core of the network. The details of these configurations will be hidden from regular end users, showing instead high-level scenarios such as pedestrian or automotive.

Apps and devices will be benchmarked using a set of KPIs, organised around 5G use cases such as videostreaming or eHealth. These KPIs will evaluate aspects such as battery or resource consumption, and QoE. To evaluate mobile devices, a set of reference mobile apps will be used. However, app developers need to provide additional information to evaluate some of these KPIs. While TRIANGLE defines generic features that will be evaluated by these KPIs, such as videostreaming, app developers need to provide how to actually use those features in their apps. This is done by providing *app user flows*, i.e. sequences of user actions that can be automatically executed on the app installed on a mobile device. Currently, app developers have to provide these app user flows that use these features individually. Figure 1 shows a high-level overview of the TRIANGLE project framework, highlighting the role of app user flows in the execution of test cases.

3 GUIDING THE GENERATION OF APP USER FLOWS

Model-based testing techniques use a model of the system under test for generating test cases with an adequate coverage. In previous work, we modelled the app under test using nested state machines [3]. By providing explicit models, a developer is able to define the realistic uses of an app, instead of generating random inputs to test it. An app state machine was composed of one or more

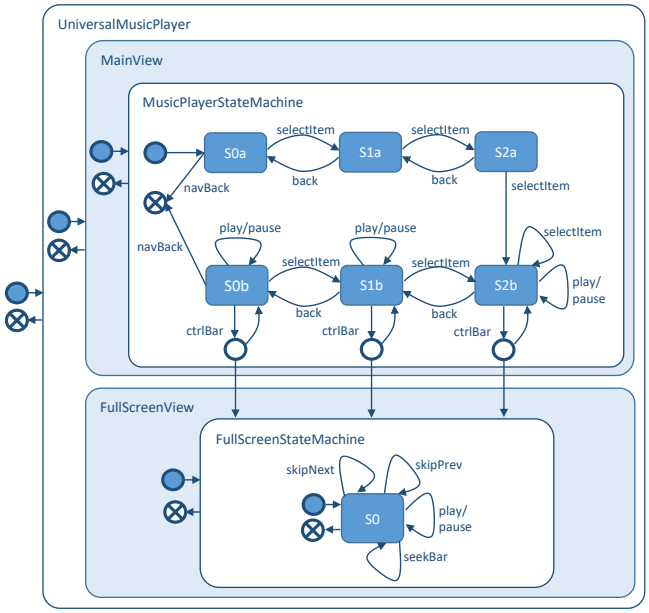
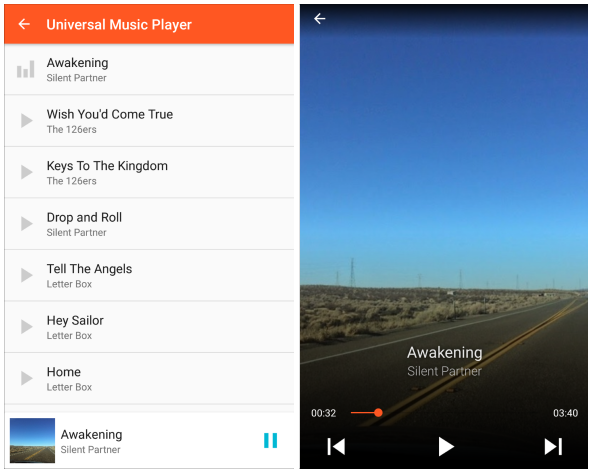


Figure 2: Universal Music Player model



(a) Main activity **(b) Full screen activity**

Figure 3: Universal Music Player GUI

view state machines, which corresponded to the different screens in an app, which in turn contained one or more state machines. The edges of a state machine represented the user actions, such as tapping a button or entering text, that should be executed when traversing the edge.

Figure 2 shows an example app model for the Universal Music Player sample app from the Android SDK, whose GUI can be seen in Figure 3. The app contains a list of songs classified by genre. The user can select the genre and the first song to play. Then, the app reproduces the list of songs in a loop starting from the selected one. The app plays music until the user exits the app or clicks the pause button. The app is divided into two activities: one for selecting a song from genre playlists, and then a full screen view with the playback controls. The first one shows the list of

```
1  DeviceType devices[DEVICES];
2  #define curBackstack devices[device].backstack
3  #define curState curBackstack.states[curBackstack.index]
4
5  proctype device_4107a7166c03af9b(int device) {
6    do
7     :: curBackstack.index > -1 &&
8      curState == St_MainView_MusicPlayerSM_S2 ->
9      // Event: selectItem
10     transition(device, VIEW_MainView, 6);
11     curState = St_MainView_MusicPlayerSM_S2b
12    :: curBackstack.index > -1 &&
13     curState == St_MainView_MusicPlayerSM_S2b ->
14     // Event: clickBack
15     transition(device, VIEW_MainView, 7);
16     curState = St_MusicPlayer_MainView_MusicPlayerSM_S1b
17    :: curBackstack.index > -1 && curState ==
         St_FullScreenView_FullScreenSM_init ->
18     pushToBackstack(device,
19       St_FullScreenView_FullScreenPlayerSM_init);
20     transition(device, VIEW_FullScreenView, 0);
21     curState = St_FullScreenPlayerView_FullScreenSM_S0
22   // ...
23   od
24 }
```

Listing 1: Extract of Promela specification for test case generation

songs and the play/pause button. The second one, provides more playback controls in full screen size. The model in Figure 2 shows the two activities (*MainView* and *FullScreenView*) and the possible user events (*Play*/*Pause*, *back*, etc.) that can happen during the app execution.

An app user flow is defined as a sequence of user events that goes from an initial state to a final state of the app state machine. Thus, by exploring the model exhaustively, we were able to generate all possible app user flows. The app model is provided as an XML file, which is translated into a Promela specification. We then used the Spin [5] model checker to explore this specification exhaustively. When a valid end state was reached, we recorded the generated app user flow in a result file. These app user flows were then converted into Java programs that performed the flows on an actual Android device, using the UiAutomator API.

Listing 1 shows part of the Promela specification generated automatically from the app model in Figure 2. The state machines are translated in a single do loop, where each branch corresponds to a transition. For instance, the one in line 7 corresponds to the transition between states S2a and S2b.

The TRIANGLE project defines the KPIs of interest that will be used to evaluate the features of mobile apps, and therefore provide the requirements for the app user flows. These requirements are specified as a set of mandatory states and/or transitions of the app model that have to be reached, along with their execution order. For instance, in audio streaming applications, the main KPIs are the bit rate (related to audio quality), the buffering time (time spent waiting until music play starts or resumes), play length (amount of data streamed) and buffering ratio (waiting time over listening time). In addition, in mobile phones, energy consumption is also relevant, especially during playback. All these KPIs require that the app starts playing music, thus a essential requirement of the app user flows is starting music playback.

Since we use the exhaustive exploration of Spin, it is natural to describe the requirements as *never claims* [5]. The never claim is a special Spin process that executes synchronously with the system model, and checks whether a property holds. If it reaches the end state (its closing curly brace), Spin states that the property is violated and produces a counter-example, which in our case is interpreted as an app user flow that satisfies the requirements. Our methodology consists of translating all requirements into a never claim to make Spin generates the app user flows that satisfy them.

Moreover, the never claim can also be used to prune the state space explored, and thus reduce the time and resources required, since Spin backtracks and explores a different execution path when the never claim is blocked.

We apply our approach to obtain the app user flows of the Universal Music Player app. In this case study, we focus on the following requirements:

- The app eventually starts playing a song, which corresponds to reach state *S2b* of the app model.
- After that it has to eventually exit. This means that the app has to pass through states *S1b*, *S0b* and the end state of the state machine.
- The full screen activity is never launched.
- The app user flow cannot execute a transition more than once.

Figure 4 represents the never claim of the case study as an automaton. The label !*fullScreen* expresses that the full screen activity has been not visited, i.e. Spin never takes the branch in line 17 in Listing 1. Labels *S2b*, !*S2b* and so on, specify that the corresponding state of the app model has been (respectively not) reached. This is checked when the branches are evaluated, e.g. in line 12. Finally, the label !*repeat* states that there are no repeated transitions. We have to define this requirement because in our Promela specification, Spin's global state contains the app user flow explored so far. Repeating a transition of the app model adds new actions to the app user flow and produces new states in Spin, thus the matching algorithm does not detect the repeated transition in the app model. Although it can seem a drawback, this behaviour allows us to describe other kinds of requirements that explicitly fire an event several times, which is very useful to discover behavioural errors of the app.

Observe that each state of the automaton has two different transitions, one that links two states, and another that loops in the same state. The linking transitions are guarded with the requirements and tracks that they are satisfied in the correct order. When the automaton end state is reached, the corresponding app user flow is returned. A looping transition lets the execution of the app model advance while its guard condition is satisfied. When none of the transitions are enabled, Spin stops exploring the current path, pruning the search state space, as commented above. Therefore, the guard of a looping transition has to be disabled when the linking transition is enabled (to correctly track the requirements) and when the current app user flow is not interesting (to prune the search). For instance, in Figure 4, the looping transitions exclude the paths that have repeated transitions or activate the full screen activity.

We have carried out some experiments using Spin 6.4.6, with two different never claims: *pr.* and *no-pr.*, as well as without one. The pr. never claim prunes the search as we have explained (see Figure 4). The *no-pr* never claim differs from *pr* in looping transitions, that

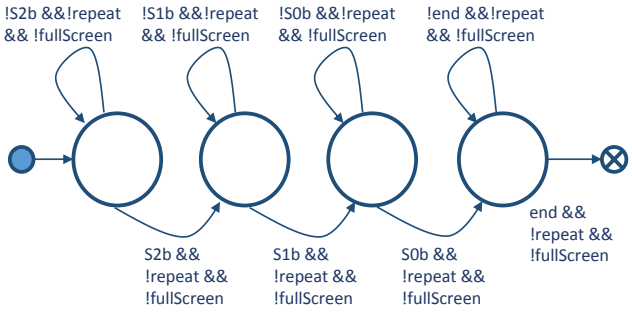


Figure 4: Pruning never claim as automaton

Table 1: App user flow generation - Experiments

Max. len.	Never	Flows	Time	Memory	States
10	pr.	18/20	<1 s	9.5 MB	1,059
10	no-pr.	18/85	55 s	11.1 MB	20,787
10	-	-/85	<1 s	10.9 MB	20,787
20	pr.	20/22	<1 s	9.6 MB	1,645
20	no-pr.	-/31,159	7.7 h[a]	1,29 GB	13,226,035
20	-	-/18,303,632	50.7 s	8,192 MB[b]	74,968,614

[a] Unfinished after 7.7 hours
[b] Unfinished after reaching memory limit of 8 GB

are guarded by *else* instead of more restrictive conditions, such as the ones in Figure 4. Table 1 shows the results. The first three rows use a maximum app user flow length (maximum number of app model transitions) of 10, and the bottom three rows use 20. The *Flows* column shows two values. The first one represents the number of app user flows that satisfy the requirements. The second one represents all the app user flows explored. Observe that for a maximum trace size of 10, the *no-pr* never claim explores 85 traces, and only 18 are app user flows that satisfy the requirements. In contrast, the pr. never claim explores 20 traces and finds the same number of valid app user flows. This means that the use of the pr. never claim drastically reduces the time elapsed in the analysis. The difference between using the *pr* and *no-pr* never claims becomes more evident when the maximum length of app user flow increases. For example, when using a maximum length of 20 and the no-pr. never claim, after more than seven hours and 31,159 different app user flows explored (most of them not satisfying the requirements), the analysis had still not finished. Therefore, our approach, which uses the *pr* never claim to prune the search state space, greatly improves the performance of test case generation process. If we do not use a never claim at all, the generation of all possible app user flows is much faster, as seen in third and sixth rows. However, the developer does not know which ones satisfy the requirements. For instance, in the sixth row, the user ends up with millions of app user flows, which is not very useful in practice. It is clear that pruning the state space using requirements is still the best option.

4 CONCLUSIONS

We have presented an approach to guide the generation app user flows (sequence of user interactions) for testing mobile apps. In our previous work, we used Spin to exhaustively explore the state space of an app model. Now, we use never claims to generate significant sequences of user interactions that allow the measurement of KPIs. In this way, we separate the app model from the requirements of the KPIs to be measured. In addition, we define pruning never claims, which drastically reduce the state space explored and the time elapsed.

The work is in an early stage, and many issues have to be addressed. First, we have to thoroughly study the KPIs and identify the mandatory requirements for the test cases. In addition, we have to define a specification language for the KPIs requirements, in such a way the developer can easily relate the mandatory requirements with the app under test, and also define new requirements. Second, we have to produce test cases with specific user input data (e.g. text input) and enriched with time information. Currently, we abstract the input data and the time elapsed between user events. Given the (possible infinite) set of user input data, we should determine some kind of equivalence relation on input data, and use only a representative subset of them. Similarly, the analysis of some KPIs may require minimum or maximum time between events. Although the view state machines can incorporate time information in transitions, it is not currently considered in the generation phase. A timing analysis can help us to diminish the app user flows to those having a minimum/maximum duration or time between events. Finally, we have to fully integrate the guided generation of app user flows in the TRIANGLE testing framework. We have to consider issues such as the (semi) automatic generation of the app model, that can be based on the analysis of the app or real user traces, and the analysis of the traces produced when the test cases are executed.

REFERENCES

[1] C. Chang and N. Lin. A constraint-based framework for test case generation in method-level black-box unit testing. *Journal of Information Science and Engineering*, 32(2):365–387, 2016.
[2] A. R. Espada, M. M. Gallardo, A. Salmerón, and P. Merino. Runtime verification of expected energy consumption in smartphones. In *Proc. of the 22nd Int. Symposium on Model Checking Software*, pages 132–149. Springer International Publishing, Aug. 2015.
[3] A. R. Espada, M. M. Gallardo, A. Salmerón, and P. Merino. Using model checking to generate test cases for android applications. In *Proc. 10th Workshop on Model Based Testing*, volume 180 of *EPTCS*, pages 7–21. Open Publishing Association, 2015.
[4] A. R. Espada, M. M. Gallardo, A. Salmerón, and P. Merino. Performance Analysis of Spotify® for Android with Model Based Testing. *Mobile Information Systems*, 2017:14, 2017.
[5] G. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, Sept. 2003.
[6] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proc. of the 5th European Dependable Computing Conference*, pages 281–292. Springer-Verlag, Apr. 2005.