Context-Dependent App Testing

Tim A. Majchrzak[1] and Matthias Schulte[2]

[1] University of Agder, Kristiansand, Norway
[2] viadee Unternehmensberatung GmbH, Münster, Germany
tima@ercis.de, Matthias.Schulte@viadee.de

Abstract. Software quality of apps often is low, which at least partly results from problems with testing them. A main problem are frequent context changes that have to be dealt with. Network parameters such as latency and usable bandwidth change while moving; usage patterns vary. To address context changes in testing, we propose a novel concept. It is based on identifying blocks of code between which context changes are possible. It helps to greatly reduce complexity.

Keywords: app, mobile, test, testing, context

1 Introduction

Mobile applications – *apps* – draw greatly from the possibilities offered by devices such as smartphones and tablets, e.g. by making use of localization via GPS. *Software testing* is a main challenge in app development. Testing software is a cumbersome task [13] and requires sophistication. App testing poses several particularities: Apps are not developed on the platform they run (mobile device) but on a PC. Testing on emulators will not yield the same results as testing natively. This also makes it laborious and hard to automate. Moreover, tool support currently is limited. Finally, many apps combine various technologies and programming languages.

Testing is greatly influenced by *context*. Mobile devices are subject to many different contexts; the simplest one is location, since mobility typically means frequent changes of position. There are many further context changes such as network condition, availability of data from sensors, and even social issues such as sharing devices. Thus, devices need to be tested taking context into considerations.

We propose a novel approach for software testing of apps that takes into account context changes. Our contributions are a sketch of context as an influencing factor of apps, the introduction of our unique approach for handling context in testing, and the demonstration of a real-life scenario to underline feasibility.

This paper is structured as follows. Section 2 highlights the relevance of context changes. Section 3 explains our approach. Application, limitations and challenges are discussed in Section 4.

2 Mobile Devices and Context

The usage paradigm of apps contrasts that of applications on PCs and that of Web sites (Webapps). The main difference is *mobility*. Even if an app not explicitly

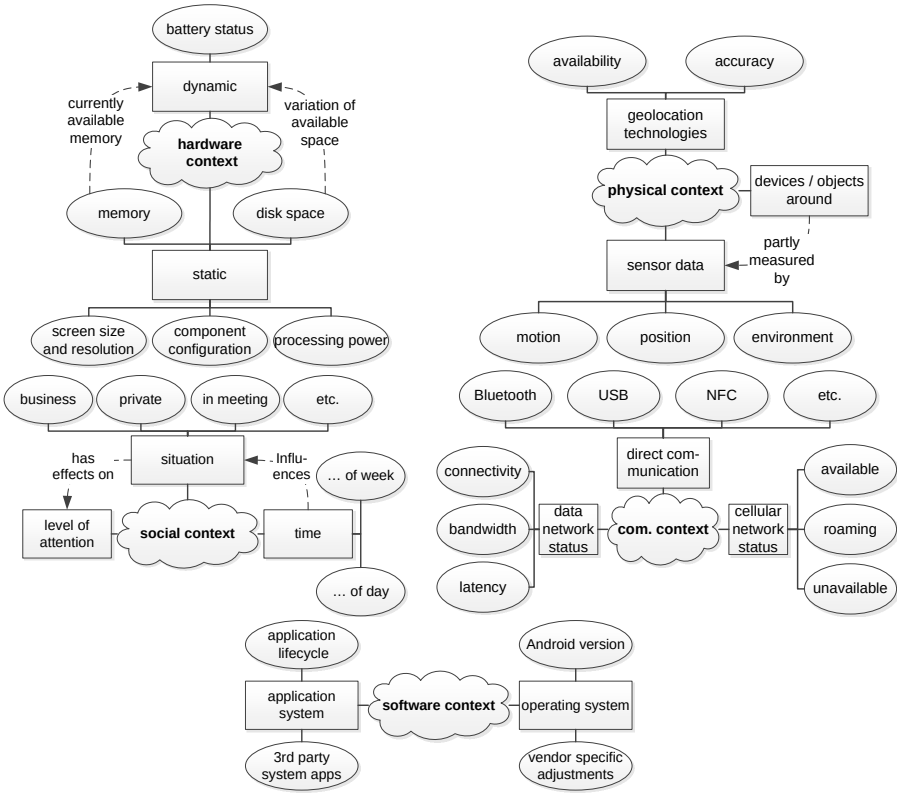

Fig. 1. Contexts

takes notice of mobility it is influenced by it due to changing conditions e.g. regarding connectivity. Apps might be halted on external events such as incoming phone calls. Moreover, they are built to be used in a changing environment.

Consequently, apps are heavily influenced by *context* but are also capable of making use of it. To give an example: when you drive out of town your smartphone might need to switch to a cellular system providing less bandwidth (adaption to context) but help you find a nearby swimming lake by matching geolocation information with map data (utilization of context).

2.1 Contexts Relevant for Mobile Devices

In the following, we propose a categorization scheme for app context that distinguishes five core contexts. It is sketched in Figure 1.

Firstly, there is the *hardware context*. Apps run on a multitude of devices, ranging from *smartwatches* to TVs. Components of devices greatly differ, leading to various screen sizes, resolutions, sensors, and input means besides touch to name just a few. Apps might not find the kind of hardware they require for proper operation and might need to adapt to hardware of different quality. Additionally, available memory and battery capacity have to be taken into consideration. The situation is worsened by the rapid progress in hardware development, making

requirements very hard to forecast. The problem can be handled by seeking for a kind of "greatest common divisor" among devices. Context testing should include various possibilities of encountered hardware, e.g. with a sensor being present or not, and considering different levels of quality (such as precision). A testing strategy similar to equivalence partitioning [13, p. 28] might be chosen.

Fragmentation in terms of hardware goes along with software fragmentation, forming the *software context*. With at least four popular platforms [9], each existing in a variety of versions and some even modified by the hardware vendors, app development is problematic. It is unlikely that soon a cross-platform approach [12] will alleviate the problem. In fact, apps existing both in a native version and as a mobile Webapp even complicate testing. Context-related problems might also arise from the combination of contextual factors of hardware and software.

Mobility means that the *physical context* is continuously changing [18]. Location determines factors such as connectivity [19]. Most devices have one or more means to determine their location. The physical context also comprises other devices that can be contacted with technology such as Bluetooth and Near Field Communication (NFC). Moreover, devices are usually equipped with a number of context-depended sensors such as gyroscopes, thermometers and similar units.

Depending on the location, connection parameters such as availability, bandwidth and latency vary. This forms the *communication context*. it is influenced by static (e.g. the carrier of a user's choice) and dynamic factors. Location typically determines which mobile services are available and which bandwidth can be used. Apps ought to be robust and maintain functionality in offline scenarios. Testing includes simulations of small bandwidths, high latencies, changes in connection quality, and abrupt unavailability of service as well as resuming service.

The *social context* is harder to grasp than the other context categories. It comprises of user-specific ways of using an app. Firstly, more and more mobile devices are used both for work and for private purposes as part of *Bring your own device* (BYOD) [7] policies. Some apps are used for work, others are used for personal reasons, and some for both (but probably used in a different way). Secondly, some devices are used by more than one person. Different people use apps uniquely despite there typical set-up single-user scenarios. Thirdly, users' attention span will not be the same in all situations. This can be explained with the mobile nature: a person that uses the smartphone while walking would risk bumping into a street light if constantly staring on the screen. The social context is very challenging for testing; its factors are fuzzy, hard to estimate and in many cases impossible to exhaustively simulate.

In consequence, app testing has to be adjusted. Frequent changes of context have to be expected and patterns of change not necessarily are predictable. The multiplicativity of contexts makes testing more complex and time-consuming.

2.2 Related Work

The relevance of context in mobile computing has been discussed as early as in 2001 [16]. Despite varying notations, context is often used in connection with awareness, i.e. devices' ability to perceive changes and react accordingly [4, 20].

Standard testing literature is valuable since Web-based applications are typically covered (e.g. [17, Chap. 22]). Techniques for testing of graphical user interfaces (GUI) can be applied to apps. Mobility is sometimes covered. Factors such as different connection speeds of mobile services [15, pp. 166] might be addressed despite not explicitly discussing context. However, textbooks on app development often do not address testing but for some exceptions such as [14].

There is a variety of – typically research-in-progress – papers on app testing. Directions of research are automation [8], user-centered testing [11], tools [10], approaches [2], and user interface testing [21, 5]. All these papers tackle testing of apps but are conceptually different to our approach. In a work complementary to ours, Amalfitano et al. [1] consider context as the result from events triggered by the user, the phone, or external activities. They propose to identify patters and use them for manual, mutation-based and exploration-based testing.

3 Context-Sensitive Testing

To combine context changes with app testing, it has to be possible to automatically change context parameters whilst test cases are executed. The naive approach would be to specify the context for each test case a priori. Asserting that a single code unit produces an expected output in a certain context would not be helpful for testing business processes, though. As many influencing contextual factors are not static but change constantly during usage, a dynamic approach is needed.

Being able to specify context changes still is static if they have to be stated *within* test cases. For scenarios including different variations of context many test cases are required, each containing the same test code. This multiplication is not desirable. Test cases would be costly to develop and hard to maintain.

3.1 General Considerations and Principles

To provide a solution that fosters dynamic changes of context, we use modularization. Test cases are split into *blocks*; between each block a *change of context* is possible. Blocks are reused and combined with context changes. Therefore, testing different scenarios is possible without duplication of test code. As blocks are the foundation of our concept, we call it *block-based context-sensitive testing*.

A given test case may result in a number of blocks, each containing *operations* and *assertions*. Similar to unit testing frameworks, operations are needed to simulate user interaction; assertions are used to verify expected behavior. Our idea is to derive blocks from existing test cases. A single case may be transformed into a structure of blocks that can be used to generate context-dependent ones. To preserve the test case's intention, blocks are ordered and executed consecutively.

Listing 1.1 illustrates in a schematic way how a test case looks like. If the test code itself would be divided into blocks, this schematic test case contains two of them: one from lines 3 to 7 (*block A*) and one from lines 10 to 12 (*block B*).

The scope of each block has to be atomic w.r.t. changing contexts. In other words, during execution of test operations belonging to one single block, the

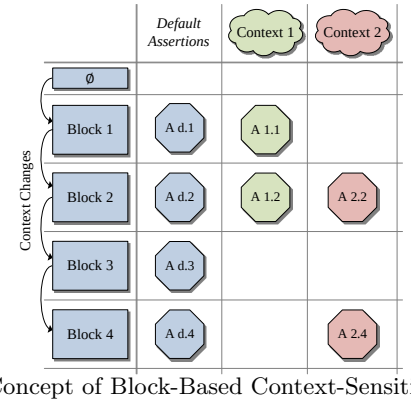

Fig. 2. Concept of Block-Based Context-Sensitive Testing

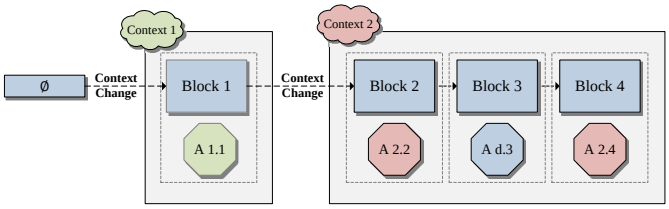

Fig. 3. Example of Test Execution Using Block-Based Context-Sensitive Testing

context remains stable. Only between blocks changes of context are possible (lines 2 and 9). To realize different scenarios, only the context changes in between have to be altered. The solution is expected to be most beneficial utilizing a generator that automatically creates test cases from blocks. Manual effort for writing context-sensitive test cases is reduced and redundant test code minimized.

```
1  public void testExample(){
2    contextChange(contextA);
3    clickSomewhere();
4    enterText();
5    clickButton();
6    ...
7    assertThat();
8
9    contextChange(contextB);
10   doSomeOtherThings();
11   ...
12   assertThat();}
```

Listing 1.1. Schematic Test Case with Context Changes

Assertions have to be extracted from blocks due to their potential dependency on a context. For each block at least one *default assertion* has to be assigned, which verifies the app's standard behavior. For each known context, a specific assertion may be defined to assess context-dependent behavior. The blocks shown in the schematic test case in Listing 1.1 therefore have to be tailored smaller. Strictly spoken, assertions in lines 7 (Block A) and 12 (Block B) are not part of the blocks but have to be treated as another building block of our concept. Depending on the app's expected behavior, assertions may be *default* or *context-sensitive*.

The building blocks of block-based context-sensitive testing are shown in Figure 2. The structure of a test case is depicted by an ordered list of blocks.

3.2 Context-Dependent Test Case Execution and Example

A possible test execution is illustrated in Figure 3. The beginning of the test case is denoted as the empty set. Before the first block is executed, `Context 1` is established. Next, an assertion fitting to the current context is searched: there is one for `Context 1`. As the assertion holds, execution continues. Between `Block 1` and `Block 2` the context is changed again. This time, `Context 2` is established and kept for the remaining execution. The second block is executed similarly to the first one. This changes when reaching `Block 3`, which does not have any specific assertions. Consequently, the expected behavior is invariant between various contexts and the default assertion is evaluated. Finally, `Block 4` is executed together with assertion `A 2.4`, which is the assigned assertion for `Context 2`.

As shown in the matrix in Figure 2, there are numerous execution paths as prior to each block the context is changeable and alternative assertions can be defined. The matrix grows with the number of contexts but typically is sparse.

To illustrate practical benefits, we implemented a proof-of-concept tool and evaluated the approach in cooperation with an industry partner. We used a simple app to explain how it works: a client for the micro blogging service *Twitter*, which allows logging in to the service and posting messages. To prepare the setting, a jar archive containing our proof-of-concept has to be added to the *classpath* of the app's Android testing project. Moreover, as the Internet connectivity has to be changed to test the app in various contexts, `ACCESS_NETWORK_STATE` and `CHANGE_NETWORK_STATE` permissions have to be added to the `AndroidManifest.xml` of the app if not already contained.

The process steps for testing are logging in with invalid credentials, logging in with correct credentials, and posting a message. Each step can be conducted in a different context. Listing 1.2 shows the first block implemented with our solution. The test operations are implemented in the operation part of the block (lines 4 to 7). The credentials are invalid: the app is expected to show a corresponding message. This is checked by the default assertion (lines 8 to 10). However, if the app is in *disconnected* context, it is expected to show an error message. This context-dependent behavior is verified by an alternative assertion (lines 14 to 16).

```
1  Context discContext = new Context(ConnectionStatus.DISCONNECTED);
2
3    Block loginIncorrectBlock = new Block(){
4     public void operation() {
5        enterText(usrName, "dummy@user.de"); enterText(pwd, "1234");
6        clickOnButton(0);
7     }
8     public void defaultAssertion() {
9       assertTrue(waitForText("Authentication failed!"));
10    }
11   };
12
13   loginIncorrectBlock.addAlternativeAssertion(discContext, new ↩
         Assertion() {
14     public void assertion() {
15       assertTrue(waitForText("Could not login: No connection."));
16     }
17   });
18 }
```

Listing 1.2. Sample Block implementation

For each of the above stated process steps a block with context-specific assertions is implemented and added to a list of blocks. A generator creates test cases as different mutations in terms of context changes from that list. In one possible test case, the first two blocks are executed in connected context and their default assertions are used for verifying. Before executing the third block, the context is changed. The context dependent assertion for that block checks whether in the *disconnected* context the message "No connection" is shown. Even in a small scenario as shown here a lot of different execution paths are possible: testing *without* our approach would be burdensome.

We used two sample generators to create test cases from the blocks explained above. In each test case the generator changes the context at different steps in the process. Finally, test cases are executed by means of JUnit. Like any other test for the Android platform, they are running on the device or emulator itself.

4 Discussion and Conclusion

In this paper we presented block-based context-sensitive testing. Our concept extends the literature of context-sensitivity in mobile computing. We have shown the viability of our approach in a case study and by presenting a prototype.

Out proof-of-concept Android tool is written in Java and can be checked out from GitHub [6]. Release under the Apache License allows free usage and modification; studying the source code will allow rapid development of similar tools for platforms such as Apple iOS. While the tool is not a core contribution of our work, it demonstrates the feasibility of our approach.

We found that by using our proof-of-concept it is possible to test apps in various contexts effectively. Blocks of test code are reusable and thus the effort in writing tests is reduced. Using generators the concept even gets more effective. Testers only have to implement blocks and define context-specific assertions once. Our approach also helps to find errors in code units where they are not expected.

However, our concept cannot (yet) be a panacea. We deem it a supportive mean for testing parts of apps which are heavily influenced by context. Due to the novelty of our approach, several limitations exist. Firstly, having a ordered linear list of blocks allows for only one way of execution, but the order of blocks and the decision if a block is executed may depend on the context used. We introduced a method for aborting test case execution to cope with this. Anyhow, our concept is not able to deal with process steps that *only* have to be executed in certain contexts. Secondly, the case study is an example of feasibility, yet not exhausting. Effectiveness has to be proven both qualitatively and quantitatively. Thirdly, our prototype is scarcely commented and not yet very user friendly.

These limitations have to be kept in mind yet do not question the general feasibility of our approach. In fact, they lead to future work. The most important task is an extension of our tool. Better support for test case generation w.r.t. context selection is desirable. Compiling best practices for context-sensitive testing would complement our work. A general challenge is to become able to

cope with ample contexts. While our approach theoretically is capable of dealing with arbitrary context changes, actually simulating them for testing is very hard.

Future work needs to refine our concept and investigate into the consequences of context changes. Moreover, we will scrutinize the relationship of our work to other methods, e.g. *data-driven testing* [3]. We will also have a look at different domains that might share properties important for testing. For example, the context-dependence described here might similarly be found in embedded systems.

Acknowledgments

We would like to thank viadee Unternehmensberatung GmbH for their support.

References

1. Amalfitano, D., Fasolino, A.R., Tramontana, P., Amatucci, N.: Considering context events in event-based testing of mobile applications. In: Proc. ICSTW. pp. 126–133. IEEE CS (2013)
2. Anand, S., Naik, M., Harrold, M.J., Yang, H.: Automated concolic testing of smartphone apps. In: Proc. ACM SIGSOFT FSE '12. pp. 59:1–59:11. ACM (2012)
3. Baker, P., Dai, Z., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C.: Data-driven testing. In: Model-Driven Testing, pp. 87–95. Springer (2008)
4. Böhmer, M., Lander, C., Krüger, A.: What's in the apps for context? In: Proc. 2013 UbiComp Adjunct Pub. pp. 1423–1426. ACM (2013)
5. Choi, W.: Automated testing of graphical user interfaces: A new algorithm and challenges. In: Proc. ACM WS on MobileDeLi. pp. 27–28. ACM (2013)
6. contextTesting @GitHub (2014), https://github.com/viadee/contextTesting
7. Disterer, G., Kleiner, C.: Using mobile devices with BYOD. Int. J. Web Portals 5(4), 33–45 (2013)
8. Gao, J., Bai, X., Tsai, W.T., Uehara, T.: Mobile application testing. Computer 47(2), 46–55 (2014)
9. Gartner Press Release (2012), http://www.gartner.com/it/page.jsp?id=1924314
10. Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: Reran: Timing- and touch-sensitive record and replay for android. In: Proc. ICSE '13. pp. 72–81. IEEE Press (2013)
11. Haller, K.: Mobile testing. SIGSOFT Softw. Eng. Notes 38(6), 1–8 (Nov 2013)
12. Heitkötter, H., Hanschke, S., Majchrzak, T.A.: Evaluating cross-platform development approaches for mobile applications. In: LNBIP, vol. 140, pp. 120–138. Springer (2013)
13. Majchrzak, T.A.: Improving Software Testing. Springer, Heidelberg (2012)
14. Milano, D.T.: Android application testing guide. Packt (2011)
15. Nguyen, H.Q.: Testing Applications on the Web. Wiley (2003)
16. Nugroho, L.E.: A context-based approach for mobile application development. Ph.D. thesis, Monash University (2001)
17. Perry, W.: Effective methods for software testing. Wiley, New York, 3rd edn. (2006)
18. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: Proc. of the 1994 1st WMCSA. pp. 85–90. IEEE CS (1994)
19. Schmidt, A., Beigl, M., Gellersen, H.W.: There is more to context than location. Computers & Graphics 23(6), 893–901 (1999)
20. Taranu, S., Tiemann, J.: General method for testing context aware applications. In: Proc. 6th Int. Workshop on MUCS. pp. 3–8. ACM (2009)
21. Yeh, C.C., Huang, S.K., Chang, S.Y.: A black-box based android GUI testing system. In: Proc. 11th MobiSys. pp. 529–530. ACM (2013)