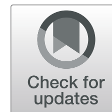
Almeida et al. Journal of the Brazilian Computer Society (2019) 25:12
https://doi.org/10.1186/s13173-019-0093-7

Journal of the Brazilian Computer Society

RESEARCH Open Access



Testing tools for Android context-aware applications: a systematic mapping

Diego R. Almeida*, Patrícia D. L. Machado and Wilkerson L. Andrade

Abstract

Context: Mobile devices, such as smartphones, have increased their capacity of information processing and sensors have been aggregated to their hardware. Such sensors allow capturing information from the environment in which they are introduced. As a result, mobile applications that use the environment and user information to provide services or perform context-based actions are increasingly common. This type of application is known as context-aware application. While software testing is an expensive activity in general, testing context-aware applications is an even more expensive and challenging activity. Thus, efforts are needed to automate testing for context-aware applications, particularly in the scope of Android, which is currently the most used operating system by smartphones.

Objective: This paper aims to identify and discuss the state-of-the-art tools that allow the automation of testing Android context-aware applications.

Method: In order to do so, we carried out a systematic mapping study (SMS) to find out the studies in the existing literature that describe or present Android testing tools. The discovered tools were then analyzed to identify their potential in testing Android context-aware applications.

Result: A total of 68 works and 80 tools were obtained as a result of the SMS. From the identified tools, five are context-aware Android application testing tools, and five are general Android application testing tools, but support the test of the context-aware feature.

Conclusion: Although context-aware application testing tools do exist, they do not support automatic generation or execution of test cases focusing on high-level contexts. Moreover, they do not support asynchronous context variations.

Keywords: Android, Context-aware application, Testing automation

Introduction

Mobile applications have become more than entertainment stuff in our lives. Such applications have become increasingly pervasive in such a way that humans are quite dependent on mobile devices and their applications. According to the research conducted by the Statistics web site portal [1], the number of mobile users can reach the five billion mark by 2019.

While mobile applications have been developed primarily for the entertainment industry, they are now touching more critical sectors such as payment systems. The exponential growth of this market and the criticality of system development demands greater attention to the reliability aspects of applications of these mobile devices. As demonstrated in some studies [2], [3], [4], mobile applications are not bug-free, and new software engineering approaches are required to test these applications [5].

Software testing is a commonly applied activity to assess whether a software behaves as expected. However, testing mobile applications can be challenging. Accordingly to Muccini et al. [4], mobile applications have a few peculiarities that make testing more difficult when compared to other kinds of computer software. Some of the peculiarities are connectivity, limited resources, autonomy, user interface, context-awareness, new operating systems updates, diversity of phones, and operating systems and touch screens.

*Correspondence: diegor@copin.ufcg.edu.br
Federal University of Campina Grande - UFCG, Aprigio Veloso Street, 882, Campina Grande, Brazil

Springer Open

© The Author(s). 2019 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Therefore, as the test difficulty increases, the demand for tools to automate the testing process of mobile applications also increases. Currently, most researchers' and practitioners' efforts in this area target the Android platform, for multiple reasons [6]: (a) At the moment, Android has the largest share of the mobile market (representing approximately 76% of the total of mobile operating system market share worldwide from June 2018 until June 2019 accordingly to the Statcounter Web Site [7]), which makes Android extremely appealing for industry practitioners; (b) as Android is installed in a range of different devices and has different releases, Android apps often suffer from cross-platform and cross-version incompatibilities, which makes manual testing of these apps particularly expensive and thus, particularly worth automating; and (c) Android is an open-source platform, which makes it a more suitable target for academic researchers, making it possible the complete access to both apps and the underlying operating system.

Nowadays, mobile devices are equipped with several sensors such as touch screen, compass, gyroscope, GPS, accelerometer, pedometer, and so on. These sensors make the development of context-aware applications possible. This paper considers *context* as any information that may characterize the situation of an entity. An entity can be defined as a person, a place, or an object that is relevant when considering the interaction between a user and an application [8]. A system is *context-aware* if it considers context information to perform its task of providing relevant information or services to a user [9]. Therefore, a context-aware application takes the information provided by the sensors to make relevant information or to direct its behavior.

This paper intends to identify tools capable of testing context-aware applications. Therefore, a systematic mapping study (SMS for short) was carried out in order to answer the following main research questions: (a) what are the Android testing tools published in the literature? And (b) what are the Android context-aware testing studies and tools published in the literature? From the answers to these research questions, we were able to analyze if the existing tools can test context-aware applications.

The SMS resulted in a total of 68 works. From them, we could see which are the research groups that work on Android application testing and which are more directly related to context awareness. Moreover, we identified 80 general Android testing tools. We analyzed these tools, identified which techniques are used mostly, which methods are implemented, which tools are available for download, and which tools are used mostly. Among the 80 general Android testing tools, we have identified five tools for testing Android context-aware applications and five tools for testing general Android applications, also allowing the testing of context-aware features. We have noticed that these tools are not able to automatically generate or execute test cases that use high-level context and that support asynchronous context variation.

The remainder of this paper is organized as follows: the "Background and related work" section presents the main concepts needed to understand this paper and the related work. The "Research method" section describes how the SMS was conducted. The "Results" section and the "Analysis and discussion" section present the results of the SMS and expose our discussions about the found tools in the context-aware Android application testing field, respectively. Finally, the "Conclusions and future works" section concludes the paper.

Background and related work

This section presents the main concepts related to this work. Also, it presents related work that addresses problems and solutions of concepts that touch the objective of this paper.

Android operating system

Android is Google's mobile operating system, and it is currently the world leader in this segment. Android is available for several platforms such as smartphones, tablets, TV (Google TV), watches (Android Wear), and glasses (Google Glass), cars (Android Auto), and it is the most widely used mobile operating system in the world.

Although Android applications are developed using the Java language, there is no Java virtual machine (JVM) in the Android operating system. In fact, until before the Android 4.4 (KitKat), what existed was a virtual machine called Dalvik, which is optimized to run on mobile devices. After that, Dalvik was replaced by ART (Android Runtime). Therefore, as soon as the bytecode (.class) is compiled, it is converted to the .dex (Dalvik Executable) format, which represents the compiled Android application. After that, the .dex files and other resources like images are compressed into a single .apk (Android Package File) file, which represents the final application. Android applications run on top of the Android framework, as can be seen in Fig. 1.

Android has a set of core apps for calendars, email, contacts, text messaging, internet browsing, and so on. Included apps have the same status of installed apps so that any app can become a default app [10].

All features of the Android OS are available to developers through APIs written in the Java language. Android APIs allow developers to reuse central and modular system components and services, making it easier to develop Android applications. Furthermore, Android provides Java framework APIs to expose the functionality of native code that requires native libraries written in C and C++. For example, a developer can manipulate 2D and 3D graphics in his/her app through the Java

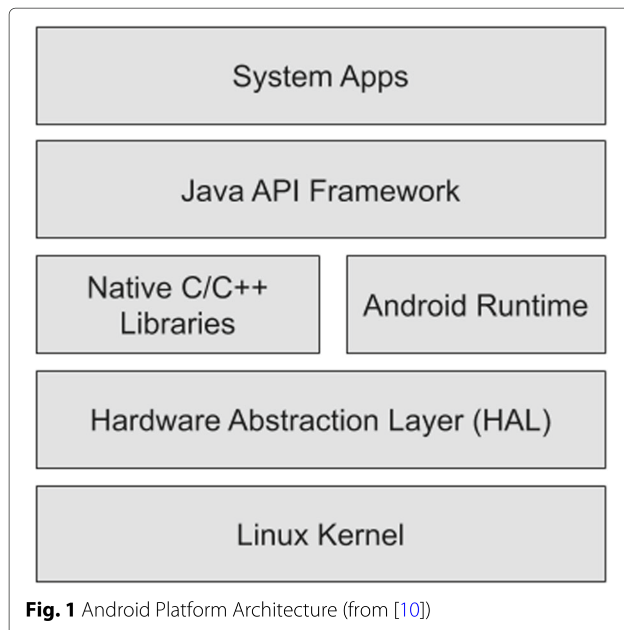


Fig. 1 Android Platform Architecture (from [10])

OpenGL API of the Android framework by accessing OpenGL ES.

As already told, from Android 4.4 (KitKat), each app runs on its process and with its instance of the Android Runtime (ART). ART is designed to run multiple virtual machines on low-memory devices by executing DEX files, a bytecode format designed especially for Android that is optimized for minimal memory footprint.

The hardware abstraction layer (HAL) provides interfaces that give access to the hardware features of the device to the higher-level Java API framework such as the camera, GPS, or Bluetooth module. When a framework API makes a call to access the device hardware, the Android system loads the library module for that hardware component.

The Android platform was built based on the Linux kernel. For example, the Android Runtime (ART) relies on the Linux kernel for underlying functionalities such as threading and low-level memory management. Another advantage of the Linux kernel is the possibility of reusing the key security features and allowing device manufacturers to develop hardware drivers for a well-known kernel.

Android applications

Initially, Android applications were written only in Java and run on the top of the Android framework presented in the "Android operating system" section. After Google I/O 2017 [11], Android applications can be written using either Java or Kotlin[1]. An android application can be composed of four component categories: (a) activity, (b) broadcast receiver, (c) content provider, and (d) service.

(a) *Activities* are focused on windows, and they are the only type of components that contain graphical user interfaces (GUI) in which the user interaction takes place; only one activity can be active at a time. Activities behavior are implemented in a .java file while the activities structure graphical interface are described in a .xml file.
(b) *Services* run in the background and can perform long-running operations, e.g., an email client may check for new mails while users are running another application. Since they do not provide a user interface, Android testing tools do not directly test them.
(c) *Content providers* are structured interfaces to shared data stores, such as contacts, photos, and calendar databases. Applications may have their own content providers and may make them available to other apps.
(d) *Broadcast receivers* handles broadcast announcements. For example, an email client may receive a notification that the battery is low and, as a result, proceed to save email drafts.

There is a mandatory XML manifest file to build Android apps that provides information regarding their life cycle management.

Although Android applications are GUI-based and mainly written in Java, they differ from Java GUI applications, particularly by the kinds of bugs they can present [2, 12]. Existing test input generation tools for Java [13–15] cannot be directly used to test Android apps, and custom tools must be adopted instead. For this reason, the academic community has made a lot of effort to research Android application testing tools and techniques. Several test input generation techniques and tools for Android applications have been proposed [16–19].

Context-aware applications

The first mobile applications had desktop application features adapted for mobile devices. Muccini et al. [4] differentiate mobile applications into two sets: (a) traditional applications that have been rewritten to run on a mobile device, which are called App4Mobile, and (b) mobile applications that use context information to generate context-based output, called MobileApps. Thus, mobile applications have become, as time goes by, increasingly pervasive. Its behavior depends not only on user inputs but also on context information. Therefore, current applications are increasingly characterized as context-aware applications.

To better understand what a context-aware application is, it is first necessary to define what context is. Some authors consider the context to be the user's environment, while others consider it to be the application's environment. Some examples are as follows:

[1] https://kotlinlang.org

- Brown [20] defines context to be the elements of the user's environment that the computer knows about;
- For Franklin and Flaschbart [21], context is the situation of the user;
- Ward et al. [22] view context as the state of the application's surroundings;
- Rodden et al. [23] define it to be the application's setting;
- Hull et al. [24] consider context as aspects of the current situation of the entire environment.

These definitions do not agree with each other and appear to be more author-specific definitions than a general definition of context. The definition most commonly accepted by several authors is the definition provided by Abowd et al.:

> Context: "any *information* that can be used to characterize the situation of *entities* (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Context is typically the location, identity and state of people, groups, and computational and physical objects." [25]

Besides being more general, this definition facilitates the understanding of what information can be considered as context information. The *entities* we identified as most relevant are places, people, and things. Places are regions of geographical space such as rooms, offices, buildings, or streets. People can be either individuals or groups, co-located or distributed. Things are either physical objects or software components and artifacts, for instance, an application or a file. The context information most commonly used by mobile applications is location, but there are several other types of context information that may be relevant to a mobile application such as temperature, brightness, time, date, mobile device tilt, geographic orientation (north, south, east, west), and so on.

Vaninha et al. [26] model context into four layers: low-level context, high-level context, situations, and scenario. In their model, context information is divided into two levels (low-level and high-level). Combinations of values of context information form situations and chronological sequences of situations form scenarios. Following the Vaninha et al. [26] model, we can divide context information into two levels:

- Low-level context: This is divided into two types:
  - Physical context: They are directly acquired from the hardware sensor measurement (e.g., location, light, sound, movement, touch, temperature or air pressure);
  - Virtual contexts: These are related to virtual data, acquired from software applications or services (for example, weather conditions, a person's mood, current speed).
- High-level context: It is the result of a combination of low-level context and virtual context processing. It uses various sources of information and combines physical and virtual sensors with additional information to solve higher tasks. For example, the "high-speed vehicle" context is an aggregation of the following low-level contexts: (a) GPS coordinates to identify at which point of a given highway the vehicle is located, b current vehicle speed, and (c) the maximum speed on that highway.

Related work

Matalonga et al. [27] performed a quasi-systematic review aiming to characterize the methods used for testing context-aware software systems (CASS). From the 1820 analyzed technical papers, 12 studies that address testing of CASS were identified. Matalonga et al. argue that, after analyzing the 12 studies, it did not found evidence that there is a complete context-aware test method for testing CASS.

Guinea et al. [28] conducted a systematic review evaluating the different phases of the software development life cycle for ubiquitous systems. The main limitations and interests of each of the phases were classified according to each phase of the development cycle. The systematic review resulted in 132 approaches addressing issues related to different phases of the software development cycle for ubiquitous systems. From the 132 approaches, only 10 are related to the test phase. Guinea et al. classified the 10 approaches by their focus or concern and obtained 5 dedicated to context-aware testing, 3 to simulators, and 2 to test adequacy. For this reason, the authors state that testing is perhaps where the results reflect that more research is needed.

Shauvik et al. [6] present a paper that performs a thorough comparison of the main existing test input generation tools for Android. In the comparison, it is evaluated the effectiveness of these tools, and their corresponding techniques, according to four metrics: (a) code coverage, b ability to detect faults, (c) ability to work on multiple platforms, and (d) ease of use. Shauvik et al. affirm Android is event-driven. Consequently, inputs are normally in the form of events, such as clicks, scrolls, and text inputs, or system events, such as the notification of newly received text messages. Thus, testing tools can generate such inputs by randomly or following a systematic exploration strategy. Considering the last case, exploration can

either be guided by a model of the app, which construction can be done statically or dynamically, or exploit techniques that aim to achieve as much code coverage as possible. Besides that, testing tools can generate events by considering Android apps as either a black box or a white box. Gray box approaches are also possible. The gray box strategy can be used to extract high-level properties of the app, such as the list of activities and the list of UI elements contained in each activity, in order to generate events that will likely expose unexplored behavior. While Shauvik et al. present unexpected results and also an analysis of weaknesses and limitations of the tools, none of the tools or any of the analyses and conclusions about the work take into account context-aware applications.

Due to the growing number of context-aware applications in recent years, many other researchers have been interested in investigating testing of such applications, [16, 17, 29, 30].

Wang et al. [17] introduced an approach to improve the test suite of a context-aware application by identifying context-aware program points where context changes may affect the application's behavior. Their approach systematically manipulates the context data fed into the application to increase its exposure to potentially valuable context variations. To do so, the technique performs the following tasks: (1) it identifies key program points where context information can effectively affect the application's behavior, (2) it generates potential variants for each existing test case that explore the execution of different context sequences, and (3) it attempts to dynamically direct application execution towards the generated context sequences.

Songhui et al. [16] investigate challenges and proposed resolutions for testing context-aware software from the perspective of four categories of challenges: context data, adequacy criteria, adaptation, and testing execution. They conclude there is no automatic testing framework that considers all of the discussed challenges, and they say they are building such a framework as an execution platform to ease the difficulty of testing context-aware software.

Thought a quasi systematic literature review, Santiago et al. [29] identified 11 relevant sources that mentioned 15 problems and 4 proposed solutions. The data were analyzed and classified into 3 groups of challenges and strategies for dealing with context-aware software systems testing.

Sanders and Walcott [30] propose an approach to test multiple states of Android devices based on existing test suites to address the testing of context-aware applications. They introduce the TADS (Testing Application to Device State) library. However, the approach currently handles only the Wi-Fi and Bluetooth sensors.

Holzmann et al. [31] and the AWARE framework[2] present very similar applications for sharing mobile context information. Holzmann et al. present an application called the "Automate toolkit" and, similar to the AWARE framework, they are able to store the usage data and context information of a device, so it is possible to store device usage scenarios such as visited screens and performed gestures. So, using scenarios that can cause failures in some application can be more carefully studied and repeated. While "Automate toolkit" emphasizes on the logging of interactions such as opened apps, visited pages, and number of interactions per page, AWARE focuses on presenting and explaining context information, that is, the main purpose of the AWARE framework is to collect data from a series of sensors on the mobile device and infer context information from them. Neither of these applications is intended to test context-aware applications, but rather to store usage scenarios.

Usman et al. [32] present a comparative study of mobile app testing approaches focusing on context events. The authors defined as comparison criteria the following six key points: events identification, method of analyzing mobile apps, the testing technique, classification of context event, validation method, and evaluation metrics. Usman et al. resemble our work on some results, such as the categorization of Android context-aware testing tools into testing approaches such as script-based testing, capture-replay, random, systematic exploration, and model-based testing. The primary search process was performed on databases and indexing system of Scopus and Google Scholar. The search was performed to return results between 2010 and 2017. In our work, a systematic mapping was performed to return results between 2008 and 2018. Therefore, we obtained a greater amount of tools and found studies. In addition, Usman et al. do not make a further reflection on the obtained results such as those we have done in our work (i.e., main authors in the field of research, key conferences, and most used tools).

Although relevant, except Usman et al. [32], none of these studies discussed so far have conducted an investigation of the ability of current testing tools for handling Android context-aware applications. Matalonga et al. [27] presented a quasi-systematic review to characterize methods but did not discuss whether the current tools are capable of testing context-aware applications. Shauvik et al. [6] perform a thorough comparison of the main current test input generation tools for Android but does not analyze context awareness features. The other authors present studies of challenges and solutions, some with tools, but with their respective limitations.

[2] http://www.awareframework.com

Research method

The purpose of an SMS is to comprehensively identify, evaluate, and interpret all work relevant to the defined research questions. Thus, this section is based on the work of Petersen et al. [33] and details the central research questions of this paper, as well as the procedure followed to identify the relevant studies required to do so.

Research questions

This systematic mapping aims at summarizing the current state of the art concerning test automation tools for Android context-aware applications. In order to do so, we conducted an SMS following the recommendations defined by Petersen et al. [33] and, therefore, proposed the following research questions (RQs):

- RQ1: What are the Android testing tools published in the literature?
  - RQ1.1: What technique do they implement?
  - RQ1.2: What are the most used ones?
- RQ2: What are the Android context-aware testing studies and tools published in the literature?
  - RQ2.1: Which research groups are involved in Android context-aware testing research?
  - RQ2.2: What are the research gaps addressed in Android context-aware testing?

For each main research question, we formulated sub-questions as listed above. Those sub-questions are answered to support our main questions. In RQ1, we aim to find out what are the existing Android testing tools currently discussed in the literature. In RQ2, we aim to identify and better understand the existing research about Android context-aware testing. To answer the research questions, we searched for studies from four digital libraries, as can be seen in the "Sources of information" section.

Sources of information

In order to gain a broad perspective, as recommended in Kitchenham's guidelines [34], we widely searched for references in electronic sources. The following databases were covered:

- ACM Digital Library[3];
- IEEE eXplore[4];
- Science Direct[5];
- Springer Link[6].

These databases cover the most relevant journals, conferences, and workshop proceedings within Software Engineering.

Search criteria

In order to select just articles related to potentially Android context-aware applications testing tools, some keywords were defined.

- *Sensibility to context*: Context-aware, context aware, context driven, context sensitive, context sensitivity, pervasive, ubiquitous, self adaptive, self adapt.
- *Others*: Android, test, testing, tool, framework.

As a result of the combination between the keywords and the connectors **AND** and **OR**, the following search string was defined:

```
''Android'' AND
(''context-aware'' OR ''context aware'' OR
 ''context driven'' OR ''context
   sensitive'' OR
 ''context sensitivity'' OR ''pervasive''
   OR
 ''ubiquitous'' OR ''self adaptive'' OR
 ''self adapt'') AND
(''test'' OR ''testing'') AND
(''tool'' OR ''framework'')
```

However, when executing the search string, the number of results was too small. Thus, we decided to split the search string into two search strings: in the first one, we would address studies related to context-aware Android applications; the second one, studies related to Android application testing tools.

Thus, the resulting search strings were the following.

Search String 1:

```
''Android'' AND
(''context-aware'' OR ''context aware'' OR
 ''context driven'' OR ''context
   sensitive'' OR
 ''context sensitivity'' OR ''pervasive''
   OR
 ''ubiquitous'' OR ''self adaptive'' OR
 ''self adapt'')
```

Search String 2:

```
''Android'' AND
(''test'' OR ''testing'') AND
(''tool'' OR ''framework'')
```

Inclusion and exclusion criteria

Studies were selected for the SMS if they met the following inclusion criteria:

1. The study describes at least one Android testing tool.
2. The study clearly describes the method and purpose of the testing tool.

[3] https://portal.acm.org
[4] https://ieeexplore.ieee.org
[5] https://www.sciencedirect.com
[6] https://link.springer.com

3. The study is written in English, Portuguese, or Spanish.

In terms of exclusion criteria, studies were excluded if they

1. Did not match the inclusion criteria.
2. Did not have relevant information about the tools.
3. Were published before 2008.
4. Described only obsolete Android testing tools.

Study selection and extraction

In order to obtain more confidence in the research results, the study selection was divided into three steps: electronic search, selection, and extraction. The electronic search was conducted executing the search strings in the sources of information. The search process performed on all databases was based on the advanced search feature provided by the online databases. The search strings were applied using an advanced command search feature and set to include meta-data of studies with initial data set up since 2008.

After executing the search strings in each of the sources of information, a total of 24,005 studies were found. The search reported too many results diverging from the objective of the research in two cases: Search String 2 of Science Direct and two Search Strings of the Springer Link. Thus, some filters were considered to obtain results closer to the objective of the work. At Science Direct, the search was filtered to find studies that presented the search string words in the title, abstract, or keywords. In Springer Link, the studies that were of the discipline of computer science and had software engineering as a subdiscipline were filtered. Thus, after applying the filters, we found 6648 as can be seen in Table 1.

Table 1 Studies found in the electronic search

		First search	Filtered search
	Search String 1	462	462
ACM	Search String 2	274	274
	Total	736	
	Search String 1	725	725
IEEE	Search String 2	267	267
	Total	992	992
	Search String 1	2284	2284
Science Direct	Search String 2	4995	294
	Total	7276	2578
	Search String 1	4062	763
Springer Link	Search String 2	10,939	1549
	Total	15,001	2342
	Total	24,005	6648

The systematic mapping was followed using the Start [35] tool. This tool was used in the organization of the systematic mapping as well as automatic identification of duplicate articles. Only the results of Springer Link could not be manipulated in the Start tool because it cannot be exported in any of the formats accepted by the tool. Thus, Microsoft Excel[7] was used. The file containing the search results in the search engines and the Start tool file containing the references can be downloaded here[8].

After reading the abstract of the 6648 studies resulted from the electronic search, a total of 143 studies were selected. The selection of these studies was carried out by reading the abstract and verifying if the study belonged to the SMS area of interest. Table 2 illustrates how many studies were accepted, rejected, and duplicated in each of the sources of information.

After the complete reading of each of the 143 selected studies, 68 studies were accepted and included in this systematic mapping study. These studies were extracted by reading each of the selected studies and noting whether they meet the inclusion and exclusion criteria of the "Inclusion and exclusion criteria" section. Figure 2 summarizes how the studies resulting from the systematic mapping of this work were selected and extracted.

Study analysis

All information extracted from the 68 found studies is presented in the "Results" section. We analyzed the number of publications per year, the number of publications per country, the main conferences in which articles were published, and the main authors in the area. The main contribution of this information was the inference of which are the groups that publish most in the area of SMS.

The "Analysis and discussion" section presents the analysis performed in the 80 tools found in the systematic mapping conducted in this work. In this analysis, we discussed which are the used testing techniques, which tools generate and/or execute test cases, test case generation strategy, which sensor data each tool considers, test approach and whether the tool is available for download. This information allowed us to answer the SMS research questions.

Validity evaluation

There are validity threats associated with all phases during the execution of this SMS. Thus, this section discusses the threats and possible mitigation strategies according to each SMS phase.

Study search and selection

We may have excluded studies during the search due to various reasons such as personal bias; this may negatively

[7] https://www.microsoft.com
[8] http://bit.ly/ArticleArchives

Table 2 Selected studies

		Quantity
ACM	Accepted	64
	Rejected	624
	Duplicated	48
IEEE	Accepted	45
	Rejected	910
	Duplicated	37
Science Direct	Accepted	13
	Rejected	2536
	Duplicated	29
Springer Link	Accepted	21
	Rejected	1855
	Duplicated	466
Total	Accepted	143
	Rejected	5925
	Duplicated	580

impact on the SMS result. The following strategies were used to reduce risks.

1. Four popular databases (e.g., IEEE Explore) on software engineering were included for the database search and we also used snowballing in the main studies;

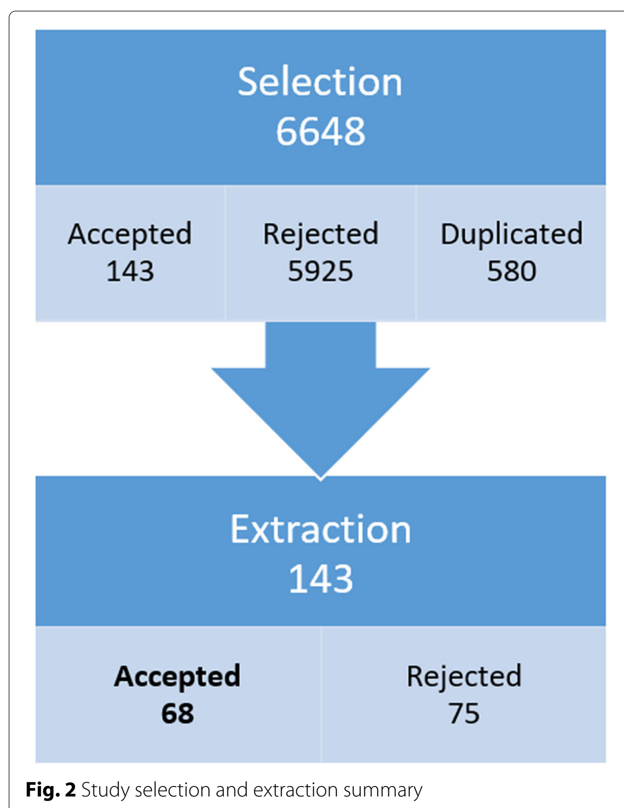


Fig. 2 Study selection and extraction summary

2. We designed and reached an agreement on inclusion and exclusion criteria (see 1) for selecting studies, which helped to avoid wrong exclusions; and
3. The electronic search and the study selection was executed twice.

Another threat is that we may have missed primary papers published before the year 2008. The Android operating system was publicly released on September 2008; since the research is directed to the Android operating system, we do not believe there are any publications in the research area before 2008.

Data extraction and analysis

Personal bias may decrease the quality of the extracted data from the studies (e.g., the incompleteness of the extracted data). The strategy used to mitigate this threat is the conduction of weakly meetings where we discussed the following:

1. Potential problems in data extraction (e.g., whether certain data should be extracted);
2. Extracted partial results;
3. Potential problems in data analysis.

Results

The systematic mapping performed resulted in the 68 studies presented in Table 3. We can see that they have been published since 2012 (Fig. 3). The largest number of studies (60.3% of the total) was published in 2014, 2015, and 2016. In 2017 and 2018, the number of publications has decreased, which supposes the beginning of disinterest in the area (Fig. 3). However, the systematic mapping focused only on studies on Android testing tools, and thus, we cannot say that the number of studies on Android testing has diminished.

From Fig. 4, the country that most published Android testing tool articles was the USA (with 35.3% of the total), followed by China, Italy, and Brazil.

Figure 5 illustrates the major conferences in which all accepted studies were published. The International Conference on Software Engineering (ICSE) is the event that most accepted studies related to our SMS, 11.8% of all of them.

Figure 6 shows the authors who most published articles in the research area of this systematic mapping. Analyzing the Google Scholar [102] profile of each of the authors, we can see that Iulian Neamtiu[9] posted 21 articles on Android application testing, Anna Rita Fasolino[10] posted 9, Domenico Amalfitano[11] posted 12, Porfirio

[9] https://scholar.google.com/citations?user=8qU-5YMAAAAJ&hl=en
[10] https://scholar.google.com/citations?user=lC5j76YAAAAJ&hl=en
[11] https://scholar.google.com/citations?user=ReafO6YAAAAJ

Table 3 Accepted studies

Studies
Imparato [36], Vieira et al. [37], Li et al. [38], Yang et al. [39], Amalfitano et al. [40] [41–43], Griebe and Gruhn [44], Bernardo et al. [45], Prathibhan et al. [46], Anand et al. [47], Zaeem et al. [48], Villanes et al. [49], Coppola et al. [50], Jensen et al. [51], Moran et al. [52], [18], Haoyin et al. [53], Wang et al. [54], Anbunathan et al. [55], Liu et al. [56], McAfee et al. [57], Nguyen et al. [58], Anbunathan et al. [59], Li et al. [19], Ye et al. [60], Jamrozik et al. [61], Machiry et al. [62], Hu et al. [63], Song et al. [64], Linares-Vasquez et al. [65], Mahmood et al. [66], Merwe et al. [67, 68], Meng et al. [69], Su et al. [70], Hu et al. [71], Choi et al. [72], Paulovsky et al. [73], Hu et al. [74], Qin et al. [75], Lin et al. [76], Wen et al. [77], Hao et al. [78], Lam et al. [79], Mirzaei et al. [80], Gomez et al. [81], Jun et al. [82], Farto et al. [83], Mao et al. [84], Neto et al. [85], Mirzaei et al. [86], Adamsen et al. [87], Salihu et al. [88], Azim et al. [89], Kaasila et al. [90], Morgado et al. [91], Zhauniarovich et al. [92], Li et al. [93], Hu et al. [94], Liu et al. [95], Hu et al. [96], Cao et al. [97], Ami et al. [98], Yan et al. [99], Chen et al. [100] and Koroglu et al. [101]

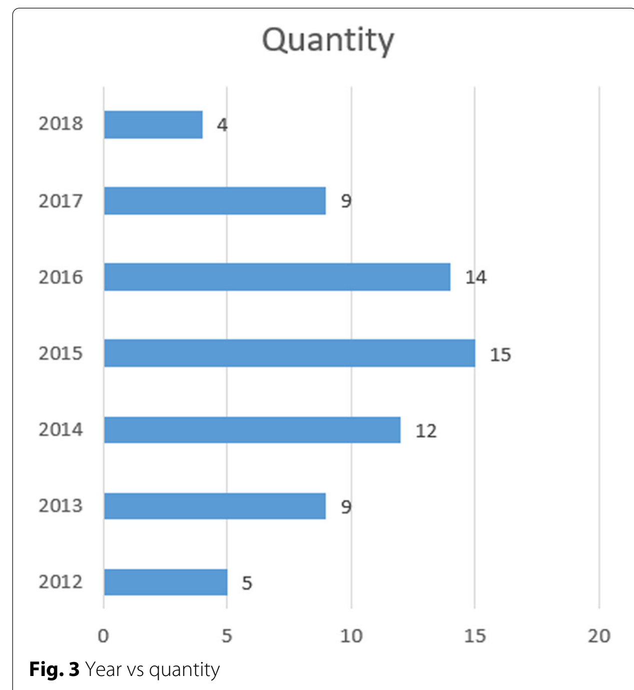


Fig. 3 Year vs quantity

Tramontana[12] posted 9, Tanzirul Azim[13] posted 11, and Yongjian Hu[14] posted 10. In addition, considering the studies accepted in this systematic mapping, we noticed that Iulian Neamtiu, Yongjian Hu, and Tanzirul Azim published 6 articles in which at least two of them wrote together and Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana published 4 articles together. Thus, from the number of articles about Android application testing and the number of articles written together, we can identify two research groups that have a significant amount of published work in the scope of this study.

Analysis and discussion

From the systematic mapping, we found 68 studies about Android testing tools. The "Results" section presented an overview of the found studies. In this section, we will answer and discuss the research questions elaborated during the systematic mapping planning.

RQ 1: What are the android testing tools published in the literature?

From the 68 found studies presented in the "Results" section, we identified 80 tools (Table 4). The tools were analyzed and classified with respect to following:

- Testing technique: The general technique the testing tool implements to test applications;
- Test case generation: Does the tool generate test cases to test the application?
- Generation strategy: If the tool generates test cases, what strategy does the tool apply?
- Use of sensor data: Does the tool consider the data from sensors to test applications?
- Test case execution: Does the tool execute test cases?
- API: Is the tool an API?
- Download availability: Is the tool currently available to download?
- Testing approach: Is the tool designed for black-box, white-box, or gray-box testing?

For the sake of space, the table with the complete classification of the attributes can be found here[15].

RQ 1.1: What technique do they implement?

Each tool tests Android applications through its implemented technique. Table 5 shows the main types of testing techniques of the identified tools. GUI testing tools are the most common ones. We found 32 tools that base their testing by identifying and exploring the interface elements to test the applications. These represent 40% of the total found tools. GUI testing tools use algorithms to identify interface elements such as text views, buttons, check boxes, and image views to generate and/or execute tests. Many tools use these graphical elements to construct state machines and thus determine the application behavioral model.

[12] https://scholar.google.com/citations?user=Q7z44GcAAAAJ&hl=en
[13] https://scholar.google.com/citations?user=YQ72v64AAAAJ&hl=en
[14] https://scholar.google.com/citations?user=gFODw24AAAAJ&hl=en
[15] http://bit.ly/Found_Tools

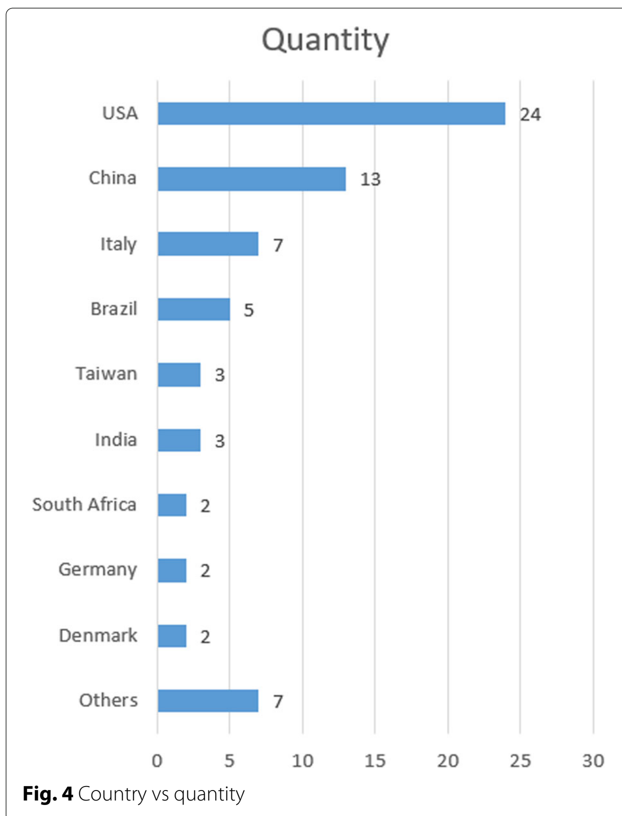


Fig. 4 Country vs quantity

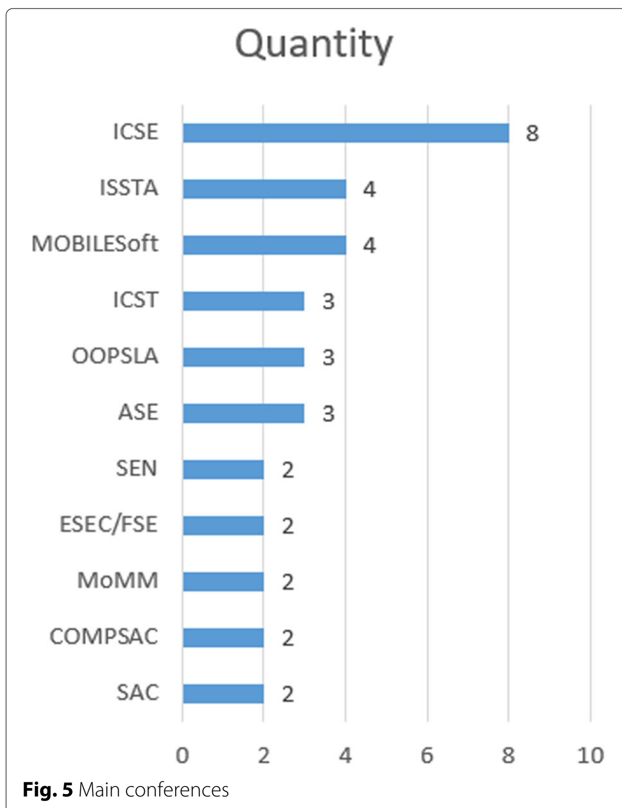


Fig. 5 Main conferences

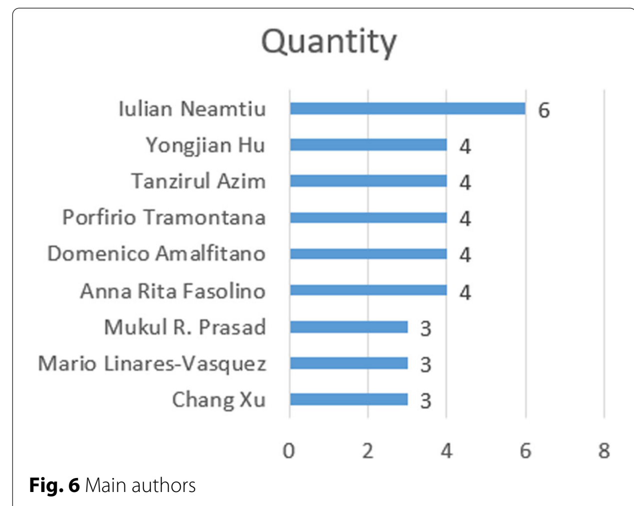


Fig. 6 Main authors

Some Android applications should be prepared to react to some events coming from the Android operating system such as low battery level, battery charging, incoming call, change of application in the foreground, airplane mode on/off, and so on. Moreover, the interaction between the user and many mobile applications does not occur exclusively through the interface elements. Many of the interactions can also be through sensors like GPS, gyroscope, compass, and accelerometer. Thus, some tools test Android applications not looking at the components of the interface, but rather through events generated for the application simulating the user touching the screen, events from the system, and also through data from the sensors of the device. For these tools, we

Table 4 Tools found in the SMS

Tools
SlumDroid [36] and GUIAnalyzer [36], Espresso [50] [45, 79], Espresso Recorder [79], UIAutomator [50], Selendroid [50], Silk Mobile [50], Sikuli GUI Automation Tool [50], Segen [85], DroidMate [61], FSMdroid [70], SwiftHand [72] [72], A^3E [89, 103], TrimDroid [80], AMOGA [88], AGRippin [40], AndroidRipper [43], Extended AndroidRipper [42], T+ [65], QUANTUM 6823880, Collider [51], EvoDroid [66], VeriDroid [95], JPF-ANDROID [68], Improved JPF-ANDROID [67], Thor [87], Monkey [104], Improved Monkey [53], PUMA [78], Dynodroid [62], ATT [69], Sapienz [84], DroidFuzzer [60], VALERA [71] [74, 79, 94, 96], MobiPlay [75] [79], RERAN [81] [79], Testdroid [90], CrashScope [18, 52], DroidBot [19], M[agi]C [58], ACTEve [47], PATS [77], BBOXTESTER [92], AppDoctor [63], ORBIT [39], Fest [45], EasyMock [45], Hamcrest [45], JUnit [45], Robolectric [45], Robotium [45] [79], Android.Test [45], DroidCrawler [54], Custom-built version of the calabash-android [44], CATE [57], MobiGUITAR [105], [41], Context Simulator [37], ADAutomation [38], MAT [46], AM-TaaS [49], VTE [55], [59], ACRT [56], EHB-Droid [64], ATG [73], SPAG-C [76], Appetizer [79], Bot-bot [79], Culebra [79], monkeyrunner [79], Mosaic [79], Ranorex [79], HiroMacro [79], Repeti-Touch [79], MAFT [82], MBTS4MA [83], SIG-Droid [86], iMPAcT [91], UGA [93], Xdroid [97], Automate toolkit [31], FragDroid [100], MobiCoMonkey [98], LAND [99], AndroFrame [101] and TCM [101]

Table 5 Types of testing technique

Technique	Quantity of tools
GUI testing	32
System events testing	17
Record and replay	14
Code coverage	6
Others	11

call them system events testing tools because they interact with the application under test by stimulating events at the system level. We identified 17 system events testing tools that represent 21% of the total found tools.

In functional testing, as important as finding a usage scenario that fails is to be able to replicate it. Replicating a failing usage scenario allows us to identify whether the application defect has been fixed. With that in mind, record and replay testing tools were developed. These tools can record a user's usage scenario and then run the same scenario as many times as the tester wishes. We identified 14 record and replay tools, representing 18% of the total found tools.

Many failures occur when executing a bug code. Consequently, the higher code coverage in a usage scenario, the greater the chance of finding bugs. For this reason, some tools test applications to maximize the amount of code covered. Among the tools found in this systematic mapping, 6 of them are code coverage tools.

Among the identified Android testing tools, 45 of them are capable of generating test cases. Each of these implements its own generation algorithms. Thus, 24 different test strategies were identified in the 45 tools that generate test cases. The most commonly used strategies are presented in Table 6.

Model-based testing (MBT) is an approach to generate test cases using a model of the application under test. In this strategy, (formal) models are used to describe the behavior of the application and, thus, generate test cases. Among the identified tools that generate test cases, we observed that 20 of them use the MBT strategy to generate their test cases. The application models mostly describe the behavior of the application under test by identifying GUI elements and changing activity based on these elements.

Table 6 Types of generation strategy

Generation strategy approach	Quantity of tools
MBT	20
GUI ripping	7
Random	6
Others	14

The second most commonly used generation strategy is the GUI ripping; it has found seven tools that implement this strategy. GUI ripping is a strategy that dynamically traverses an app's GUI and, based on its GUI elements, creates its state-machine model or a tree graph.

The third most used strategy by the found tools was the random strategy. A total of six tools implements a random strategy. Although it is a less ingenious strategy than the others, some studies point out that it is a very efficient strategy to find crashes [6].

Regarding the remaining data acquired from the questions presented at the beginning of this section about the characteristics of the tools (use of sensor data, test case generation, test case execution, download availability, and testing approach), Table 7 summarizes how many tools have each of them.

RQ 1.2: What are the most used ones?

Among the studies found by the systematic mapping, Bernardo et al. [45] present an investigation on 19 open-source mobile applications for Android in order to identify how automated tests are employed in practice. They concluded that 47% of these applications have some kind of automated tests, and they observed that the most used testing tools were JUnit, Android.Test, Hamcrest, Robolectric, EasyMock, Robotium, and Fest. Finally, Bernardo et al. observed that the most important challenges in testing Android applications such as rich GUIs, limited resources, and sensors have not been properly handled in the automated tests of the analyzed 19 open-source applications.

Linares-Vásquez et al. [106] conducted a survey on 102 Android mobile developers about their practices when performing testing. One of the questions to be answered by the survey was "What tools do you use for automated testing?". As a result, Linares-Vásquez et al. concluded that "The most used tool is JUnit (45 participants), followed by Roboelectic with 16 answers, and Robotium with 11 answers. 28 participants explicitly mentioned they have not used any automated tool for testing mobile apps. 39 out of 55 tools were mentioned only by one participant each, which suggests that mobile developers do not use a well-established set of tools for automated testing."

Villanes et al. [107] performed a study using the Stack Overflow[16] with the intention to analyze and cluster the main topics of interest on Android testing. One of their results pointed out that recently, developers have shown increased interest in the Appium[17], Espresso, Monkey, and Robotium tools.

[16]https://stackoverflow.com
[17]http://appium.io

Table 7 Number of tools that presents a given characteristic

Characteristic	Quantity of tools
Use of sensor data	9
Test case generation	45
Test case execution	75
Download availability	43
White box	14
Gray box	7
Black box	59

Table 8 Most cited study and tools

		Citations		
	Tool	ACM	IEEE	Scholar
Machiry et al. [62]	Dynodroid	119	0	395
Amalfitano et al. [43]	AndroidRipper	88	94	356
Anand et al. [47]	ACTEve	80	0	292
Azim et al. [89]	A3E	88	0	283
Choi et al. [72]	SwiftHand	63	0	233
Gomez et al. [81]	RERAN	75	60	227
Yang et al. [39]	ORBIT	60	0	218
Hao et al. [78]	PUMA	69	0	183
Jensen et al. [51]	Collider	52	0	155
Amalfitano et al. [41]	MobiGUITAR	0	51	148

Bernardo et al. did not present a significant amount of applications when compared to Vasquez et al. and Villanes et al. work. Thus, we can say, based on these studies, with greater certainty that the Robotium, JUnit, Roboelectic, Appium, Espresso, and Monkey tools are, according to Vasquez et al. and Villanes et al., the most used tools for testing Android applications.

In addition, we observed which of the studies identified in the SMS are the most cited in ACM, IEEE, and Google Scholar. Table 8 presents the studies and their respective tools that are most cited among the identified studies and tools.

RQ 2: What are the android context-aware testing studies and tools published in the literature?

From the 68 selected studies, Griebe and Gruhn [44], Vieira et al. [37], and Amalfitano et al. [42] explicitly focus on testing of context-aware applications. The "Custom-built version of the Calabash-Android" section, "Context simulator" section. and "Extended AndroidRipper" section discuss the found Android context-aware application testing tools investigated in these three studies. Besides that, these three studies cite the other two tools that also explicitly test context-aware applications: ContextDrive and TestAWARE. These tools are discussed in the "ContextDrive" section and the "TestAWARE" section, respectively.

Custom-built version of the Calabash-Android

Griebe and Gruhn [44] propose a model-based approach to improve the testing of context-aware mobile applications. Their approach is based on a four-tier process system as follows:

- Tier 1: UML activity diagrams models are enriched with context information using a UML profile developed for integrating context information into UML models;
- Tier 2: Models are then transformed into Petri Nets for analyzing and processing structural model properties (e.g., parallel or cyclic control flows);
- Tier 3: From the Petri Net representation, a platform and technology-independent system testing model is generated that includes context information relevant for the test case execution; and
- Tier 4: Platform and technology-specific test cases are generated that can be executed using platform-specific automation technology (e.g., JUnit, Calabash-Android/iOS, Robotium).

To assess the proposed approach, Griebe and Gruhn have extended the Calabash[18] tool to implement it. Calabash is a test automation framework that supports the creation and execution of automated acceptance tests for Android and iOS apps without the necessity of coding skills [108]. It works by enabling automatic UI interactions within an application such as pressing buttons, inputting text, validating responses, and so on.

Calabash is a completely free and open-source tool. It uses the Gherkin pattern. Gherkin is a writing pattern for an executable specification that, through keywords, maintains a standard for the writing of execution criteria called Given, When, and Then. In order to do so, Calabash expresses the test cases as cucumber features [109].

A limitation of the Griebe and Gruhn approach is the need for creating a model that describes possible AUT activities. Modeling is not a widely understood activity between testers and developers, and a poorly designed model can lead to false positives or false negatives in test verdicts.

Context simulator

Vieira et al. [37] argue that testing context-aware applications in the lab is difficult because of the number of different scenarios and situations that a user might

[18] https://github.com/calabash/calabash-android

be involved with. Hence, the Android platform provides simulation tools to support the physical sensor test. However, it is not enough to test context-aware applications only at the physical sensors level. Considering that, Vieira et al. have developed a simulator that simulates a real laboratory environment. The simulator provides support for modeling and simulation of context in different levels: physical and logical context, situations, and scenarios.

The simulation is separated into two main components: the desktop application and the mobile component:

- The desktop application: it is responsible for context modeling, simulation execution, and context transmission to the mobile device
- The mobile component: it receives signals from the desktop application and processes the data. The mobile component is responsible for simulating context data and for examining the reaction of the app under the simulated context

The modeling in the context simulator is made in four different levels:

1. Low-level context: the data can be acquired from hardware sensor measuring (e.g., location, light, movement or touch), named as physical context, or the data can be acquired from software applications or services (e.g., current activity of an employee determined by his calendar), named as virtual context;
2. High-level context (or logical context): the combination of low-level context and virtual context processing results in a high-level context. For example, a context “Room 001 at Fraunhofer” is identified through an aggregation of two low-level sources: GPS coordinates from “Fraunhofer” and Wi-Fi identification of “Room 001”;
3. Situation: it is the composition of high-level contexts. The situation represents the circumstances in which someone currently is. For example, the situation “Meeting 12–13 at Room 001 at Fraunhofer” is a situation composed by three high-level contexts: “Meeting” (it can be a specific date and time plus an appointment in the user’s calendar), “Room 001”, and “Fraunhofer”; and
4. Scenario: a scenario is a chain of situations for causal relations. In other words, a scenario is a time-ordered sequence of situations.

The context simulator supports a large variety of context sources, 22 contexts divided into 6 categories supporting 41 context sources [37].

A limitation of the context simulator is that the tester must model each test case. That is, if the tester wishes to test an AUT under possible adverse situations such as weak GPS signal, receiving a phone call and changing Internet connection conditions, then the tester should model all scenarios that he/she wishes to test.

Extended AndroidRipper

Amalfitano et al. [42] analyzed bug reports from open-source applications available at GitHub[19] and Google Code[20]. From the results, they defined some use scenarios, called by them as event-patterns, that represent a use case which presents more potential to failure in context-aware applications. Some examples of event-patterns are as follows:

- Loss and successive recovery of GPS signal while walking;
- Network instability;
- The user enables the GPS provider through the settings menu and starts walking; and
- Incoming of a phone call after any other event.

Amalfitano et al. carried out an experiment with the objective of examining if, in fact, the event-patterns represent scenarios of a greater chance of context-aware application failures. Thus, they extended the tool AndroidRipper [43]. The extended AndroidRipper is able to fire context events such as location changes, enabling/disabling of GPS, changes in orientation, acceleration changes, reception of text messages and phone calls, and shooting of photos with the camera. Both versions of AndroidRipper explore the application under test looking for crashes measuring the obtained code coverage and automatically generating Android JUnit test cases that reproduce the explored executions.

The extended AndroidRipper tool generates test cases watching for events that cause a reaction from the application. Once the events that cause a reaction are detected, the technique of Amalfitano et al. generates test cases based on event-patterns identified by the authors. Therefore, the tool does not focus on testing high-level context variations.

Other tools from cited papers

By studying the papers of Griebe and Gruhn [44], Vieira et al. [37], and Amalfitano et al. [42], we found two papers related to testing context-aware Android applications: Mirza and Khan [110] and Luo et al. [111]. The corresponding tools are presented in the sequel.

ContextDrive Mirza and Khan [110] argues that testing context-aware applications is a difficult task due to challenges such as developing test adequacy and coverage criteria, context adaptation, context data generation,

[19] https://github.com
[20] http://code.google.com

designing context-aware test cases, developing test oracle, and devising new testing techniques to test context-aware applications. In response to these challenges, they argue that context adaptation cannot be modeled using a standard notation such as the UML activity diagram. Therefore, Mirza and Khan extended the UML activity diagram, by adding a context-aware activity node, for behavior modeling of context-aware applications.

Mirza and Khan proposed a test automation framework named as ContextDrive. Its proposed model consists of six phases.

1. First phase: An UML activity diagram is used to model the application under test. In this phase, it is proposed a new element for the UML activity diagram for modeling context-aware applications;
2. Second phase: The UML activity diagram is transformed into a testing model;
3. Third phase: The test model is annotated in order to enhance readability and maintainability;
4. Fourth: Abstract test cases are generated;
5. Fifth: Abstract test cases are converted into platform-specific executable test scripts; and
6. Sixth: The test scripts are executed.

Mirza and Khan's technique is similar to the one implemented in the tool of the "Custom-built version of the Calabash-Android" section. Therefore, there is also the restriction that the tester has experience in UML activity diagram modeling. In addition, the tool uses static data to execute test cases. Therefore, testing situations that use a lot of sensor data becomes infeasible (i.e., testing a GPS navigator application).

TestAWARE One of the difficulties in testing context-aware applications is the heterogeneity of context information and the difficulty and/or high cost of reproducing contextual settings. As an example, Luo et al. [111] present a real-time fall detection application; the application detects when the user drops the mobile phone under different circumstances such as falling out of the pocket or falling out of the hand. The application is programmed to send an email to a caregiver every time a fall event is detected by the phone. For this application, testing new versions of the application is very costly. Thus, Luo et al. [111] introduce the TestAWARE tool.

TestAWARE is able to download, replay, and emulate contextual data on either physical or emulators devices. In other words, the tool is able to obtain and replay "context" and thus provide a reliable and repeatable setting for testing context-aware applications.

Luo et al. compare their tools with other available tools. In summary, they say TestAWARE aims at a wide variety of mobile context-aware applications and testing scenarios. It is possible because TestAWARE incorporates heterogeneous data (i.e., sensory data, events, and audio), multiple data sources (i.e., online, local, and manipulated data), black-box and white-box testing, functional/non-functional property examination, and the environments of device/emulator.

A limitation of the tool is that it is not possible to create test cases without executing each test case at least once in a real device in the real scenario. That is because it is a record and replay tool, it is first necessary to record the test cases and, therefore, it is necessary to submit the AUT on a real device under each of the conditions to be tested.

Potential tools for testing context-aware applications

Among the found studies, Moran et al. [18, 52], Qin et al. [75], Yongjian and Iulian [71], Gomez et al. [81], and Farto et al. [83] present tools that were not intended for context-aware application testing. However, they support the testing of context-aware features.

CrashScope Moran et al. [18, 52] argue that one of the most difficult and important maintenance tasks is the creation and resolution of bug reports. For this reason, they introduced the CrashScope tool. The tool is capable of generating augmented crash reports with screenshots, crash reproduction steps, and captured exception stack trace, along with a script to reproduce the crash on a target device. In order to do so, CrashScope explores the application under test by performing input generation by static and dynamic analyses which include automatic text generation capabilities based on context information such as device orientation, wireless interfaces, and sensors data.

The CrashScope GUI ripping engine systematically executes the application under test using various strategies. Then, the tool first checks for contextual features that should be tested according to the exploration strategy. So, the GUI ripping engine checks if the current activity is suitable for exercising a particular contextual feature in adverse conditions. The testing of contextual features in adverse conditions consists in setting unexpected values to the sensors (GPS, accelerometer, etc.) that would not typically be possible under normal conditions. For instance, to test the GPS in an adverse contextual condition, CrashScope sets the value to coordinates that do not represent physical GPS coordinates. In other words, for each running Activity, CrashScope checks what are the possible contextual features, checks if contextual features should be enabled / disabled, and sets feature values. CrashScope attempts to produce crashes by disabling and enabling sensors as well as sending unexpected (e.g., highly unfeasible) values. Because of that, there are scenarios that cannot be tested in CrashScope (i.e., testing if the application crashes if the user leaves a pre-established route).

MobiPlay Accordingly to Qin et. al. [75], MobiPlay is the first record and replay tool that is able to capture all possible inputs at the application layer, that is, MobiPlay is the first tool capable of recording and replaying, at the application layer, all the interactions between the Android app and both the user and the environment the mobile phone is inserted into.

While the user is executing the app, MobiPlay records every input the application receives and the interval time between every two consecutive inputs. After that, the tool can re-execute the application under test with the same provided inputs when executing it. The expected result is that the application behaves exactly the same way as the original execution. Basically, MobiPlay is composed of two components: a mobile phone and a remote server. Initially, the mobile phone sends and saves all sensor data and user interactions to the remote server that also stores it. From there, the remote server can reproduce the executed scenario by sending back the saved data to the mobile phone.

The application under test is called target app, and it is installed at the remote server, not at the mobile phone. The communication between the mobile phone and the target app will be done through the client app. The client app is installed at the mobile phone and it is a typical Android app that does not require root privilege and is dedicated to intercepting all the input data for the target app. The basic idea of MobiPlay is that the target app actually runs on the server, while the user interacts with the client app on the mobile phone in a way that the user is not explicitly aware that he is, in effect, using a thin client. The client app shows the GUI of the target app in real-time on the mobile phone just like the way as if the target app was actually running on the mobile phone. While the user interacts with the target app through the client app, the server records all the touch screen gestures (pinch, swipe, click, long click, multi touches, and so on) and the other inputs provided by the sensors like gyroscope, compass, GPS, and so on, in a transparent way to the user. Once the inputs are recorded, MobiPlay can re-execute the target app with the same inputs and at the same interval time, simulating the interaction between the user and the target app.

Just like the TestAWARE tool (the "TestAWARE" section), MobiPlay first needs to record the test cases that the tester wants to check.

VALERA *V*ers*A*tile yet *L*ightweight r*E*cord and *R*eplay for *A*ndroid (VALERA) is a tool capable of record and replay Android apps by focusing on sensors and event streams, rather than system calls or the stream instruction. Its approach promises to be effective yet lightweight. VALERA is able to record and replay inputs from the network, GPS, camera, microphone, touchscreen, accelerometer, compass, and other apps via IPC. The main concern of the authors is to be able to record and replay Android applications with minimal overhead. Therefore, they claim to be able to maintain performance overhead low, on average 1.01% for record and 1.02% for replay. The timing overhead is very important when replaying an application. The variation of the original time of the application data entries can cause different behavior than when recording the iteration data with the application. For this reason, VALERA is designed to minimize timing overhead. In order to evaluate VALERA, the tool was exercised against 50 applications with different sensors. The evaluation consisted in exercising the relevant sensors of each application, e.g., scanning a barcode for the Barcode Scanner, Amazon Mobile, and Walmart apps; playing a song externally so that apps Shazam, Tune Wiki, or SoundCloud would attempt to recognize it; driving a car to record a navigation route for Waze, GPSNavig.&Maps, and NavFreeUSA; and so on.

VALERA has the same limitations as TestAWARE and MobiPlay.

RERAN It is a black-box record and replay tool capable of capturing the low-level event stream on the phone, which includes both GUI events and sensor events, and replaying it with microsecond accuracy. RERAN is a previous record and replay system of the authors of VALERA. It is similar to VALERA but with some limitations. RERAN is unable to replay sensors whose events are made available to applications through system services rather than through the low-level event interface (e.g., camera and GPS). When validating the tool, the authors declare RERAN was able to record and replay 86 out of the Top-100 Android apps on Google Play and to reproduce bugs in popular apps, e.g., Firefox, Facebook, and Quickoffice.

RERAN has the same limitations as TestAWARE, MobiPlay, and VALERA. Another limitation of RERAN is that it does not support testing the GPS sensor.

MBTS4MA Farto et al. [83] proposed an MBT approach for modeling mobile apps in which test models are reused to reduce the effort on concretization and verify other characteristics such as device-specific events, unpredictable users' interaction, telephony events for GSM/text messages, and sensors and hardware events.

The approach is based on an MBT process with Event Sequence Graphs (ESGs) models representing the features of a mobile app under test. Specifically, the models are focused on system testing, mainly user's and GUI's events. Farto et al. implemented the proposed testing approach in a tool called MBTS4MA (Model-Based Test Suite For Mobile Apps).

MBTS4MA provides a GUI for modeling. Thus, it supports the design of ESG models integrated with the mobile

app data like labels, activity names, and general configurations. Although the models are focused on system testing, mainly user's and GUI's events, it is also possible to test sensors and hardware events. The supported sensor events are change acceleration data, change GPS data, disable Bluetooth, enable Bluetooth, and update coordinates. However, the authors argue that it is possible to extend the stereotypes of the tool to support more sensor events.

Just like the custom-built version of the Calabash-Android tool (the "Custom-built version of the Calabash-Android" section), MBTS4MA needs the creation of a model that represents the features of a mobile app under test.

RQ 2.1: Which research groups are involved in android context-aware testing research?

In order to answer this research question, we have observed the publications of the authors of the Android context-aware testing studies, such as Griebe and Gruhn [44], Vieira et al. [37], and Amalfitano et al. [42].

The authors of Griebe and Gruhn [44] are Tobias Griebe[21] and Volker Gruhn[22]. Both authors have written only two more publications that refer to context-aware applications:

- "Towards Automated UI-Tests for Sensor-Based Mobile Applications" [112]: presents an approach that integrates sensor information into UI acceptance testing. The approach uses a sensor simulation engine to execute test cases automatically.
- "A Framework for Building and Operating Context-Aware Mobile Applications" [113]: presents a work-in-progress paper with the description of a framework architecture design to address the following context-aware mobile applications problems: interoperability, dynamic adaptability, and context handling in a frequently changing environment.

The authors of Vieira et al. [37] are Vaninha Vieira[23], Konstantin Holl[24], and Michael Hassel[25]. Vaninha Vieira is a professor of Computer Science at Federal University of Bahia, Brazil. Her research interests include context-aware computing, mobile and ubiquitous computing, collaborative systems and crowdsourcing, gamification and user engagement, and smart cities (crisis and emergency management, intelligent transportation systems). Among her publications, we can note the interest in mobile applications concerning to context modeling, quality assurance, context-sensitive systems development, context management, and so on. Konstantin Holl has published papers related to quality assurance, but nothing can be seen about the research interest of Michael Hassel due to the lack of publications.

The authors of Amalfitano et al. [42] are Domenico Amalfitano[26], Anna Rita Fasolino[27], Porfirio Tramontana[28], and Nicola Amatucci[29]. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana are not only professors of the same institution (University of Naples Federico II) but also most of their articles were written together. Their publication concerns software engineering, testing, and reverse engineering. Many of the testing publications are about Android app testing. In particular, they have a lot of experience in the GUI ripping technique. Most of Nicola Amatucci's publications are about testing on Android applications. Most of them written together with Domenico Amalfitano, Anna Rita Fasolino, or Porfirio Tramontana.

All of these authors have significant publications regarding mobile application testing. Besides them, as mentioned in the "Results" section, we can refer to Iulian Neamtiu, Tanzirul Azim, and Yongjian Hu who have great contributions in the research area. However, among the studied authors, Vaninha Vieira is the author who most directly contributed to the research on context-aware applications.

RQ 2.2: What are the research gaps addressed in android context-aware testing?

In this paper, we identify five tools for testing of context-aware Android applications and five tools that support testing of context-aware applications, totaling 10 tools.

The context-aware application testing has challenges such as a wide variety of context data types and context variation. There is a huge variety of context data types. The most commonly used context data type is location, acquired by the GPS sensor. However, there are many other types of data, such as temperature, orientation, brightness, time, and date.

Context-aware applications use context data provided by sensors to provide service or information. Waze[30], for example, uses the GPS, the time, and information provided by the cloud to inform the driver about obstacles along the way to the final destination. However, many context-aware applications use combinations of sensor information to infer contexts and, from these inferred contexts, provide service or information. Vieira et al. [37] call low-level context the context information that is directly collected from sensors or from other sources

[21] https://dblp.org/pers/hd/g/Griebe:Tobias
[22] https://dblp.uni-trier.de/pers/hd/g/Gruhn:Volker
[23] https://scholar.google.com/citations?user=tkNSlXIAAAAJ&hl
[24] https://dblp.org/pers/hd/h/Holl:Konstantin
[25] https://dblp.org/pers/hd/h/Hassel:Michael
[26] https://scholar.google.com/citations?user=ReafO6YAAAAJ&hl=en
[27] https://scholar.google.com/citations?user=lC5j76YAAAAJ&hl=es
[28] https://scholar.google.com/citations?user=Q7z44GcAAAAJ&hl=en
[29] https://scholar.google.it/citations?user=AaLCcaAAAAAJ&hl=it
[30] https://www.waze.com

of information such as database or cloud, and high-level context for the contexts that are the product of the combination of low-level contexts.

Many context-aware applications use high-level context to provide their services or information. Samsung has developed an application called Samsung Health [114] that tracks user's physical activities. Combined with its Smart Watch, the application monitors heartbeat, movement, steps, geographical location, time of day, and other information. From this information, the application infers contexts in which the user is and then concludes whether the user is practicing physical activity or whether he is at rest. Taking the example of the Samsung Health application, Table 9 exposes some examples of high-level contexts from the composition of low-level contexts.

As we have said, another challenge in testing context-aware applications is the constant variation of context. The context changes asynchronously and the application must respond correctly and effectively to context variations. Taking Samsung Health as an example, the application must realize when the user is changing their activities throughout the day and thus provides all the information and services in the correct way. Thus, when the user is sleeping and getting up, the application should stop counting the time and the quality of sleep. If the user starts walking, the application must count for time, distance, and lost calories. When the user stops walking and gets in the car and drives to home, the application should stop counting the walking information and understand that the user is at rest, even though he is moving.

Table 9 High-level context examples

Brief description	Low-level contexts	High-level context
If the user has stopped for more than 1 h since it is not at night, the application infers that the user is at rest for a long time and suggests that the user take a short walk or lengthen.	Time, GPS, pedometer accelerometer, heartbeat	Long rest
If the user is in full rest with low heart rate, the application infers that the user is sleeping and counts the duration of sleep as well as infers the quality of sleep based on the luminance, noise and amount of movements that the user makes while sleeping.	Time, GPS, pedometer accelerometer, brightness, noise, heartbeat	Sleeping
If the user is walking, the application monitors the distance and speed. From this information and the user's profile (weight and age) the application infers the amount of lost calories.	GPS, pedometer, age, weight	Walking
If the user is pedaling, the application infers that the user is riding a bike and then calculates the time, distance and lost calories.	GPS, pedometer accelerometer, noise, heartbeat	Riding a bike

Considering the difficulties of testing context-aware applications, the 10 tools identified in this work were analyzed and compared according to 11 questions raised:

- Q1: What low-level context data does the tool support?
- Q2: Does the tool support high-level context data?
- Q3: Are context data treated differently?
- Q4: Is it possible to test context variations?
- Q5: Is it possible to test abnormal context situations?
- Q6: What criteria is used to select the context data?
- Q7: What is the test stop criterion?
- Q8: Does the tool generate test cases?
- Q9: Is the tool white box, black box, or gray box?
- Q10: Does it need instrumentation in the code?
- Q11: Is the tool automatic or semi-automatic?

Table 10 presents the result of the analysis of the 10 tools by looking at the 11 questions.

The first observation we had of the tools was on the type of context data they support. With the exception of RERAN, they all support GPS. It was natural to expect this result since location is the most commonly used data type by context-aware applications. We can also see that Context Simulator and ContextDrive are the only tools that support all low-level context data types. In addition, these are the only tools that support high-level context data.

Mirza and Khan [110] propose an extension of the UML activity diagram for modeling high-level context variation. Thus, their ContextDrive tool can test variations from one context to another. Authors use static data to execute the test cases. Therefore, the tool is unable to generate new test cases automatically.

The Context Simulator tool provides a graphical interface for the tester that enables the creation of application usage scenarios. Thus, it is possible for the tester to simulate high-level contexts. To do this, the tester explicitly describes each test case he/she wants to execute as well as which sensor values are going to be used in the test.

Context variations occur asynchronously and some of them in a totally unexpected way. When using a context-aware application, a phone call can be received and, during the calling, the user context may change. As another example, it is possible for the GPS signal to drop and then return after a few moments.

Although three tools support context variation testing, none of them is able to automatically generate test cases that use high-level contexts and test variations of high-level contexts, taking into account unexpected scenarios such as the event-patterns described by Amalfitano et al. [42], presented in the "Extended AndroidRipper" section.

Table 10 Tools comparison

	Tool									
	Custom Calabash [44]	Context simulator [37]	Extended AndroidRipper [42]	CrashScope [18]	MobiPlay [75, 79]	VALERA [71, 74, 79, 94, 96]	RERAN [79, 81]	MBTS4MA [83]	ContextDrive [110]	TestAWARE [111]
Q1										
GPS	X	X	X	X	X	X		X	X	X
Wi-Fi	X	X		X		X		X	X	X
Accelerometer	X	X		X	X	X	X	X	X	X
Thermometer	X	X	X	X				X	X	X
Barometer		X	X		X			X	X	X
Light-sensor	X	X			X		X		X	X
Magnetometer	X	X	X	X	X	X	X		X	X
Gyroscope		X			X			X	X	X
Clock		X							X	
Calendar		X							X	
Other		Camera, microphone, battery level, call, text message, alarm, etc.	Call, text message, battery level, USB, etc.			Microphone		Bluetooth, Call, text message		
Q2	No	Yes	No	No	No	No	No	No	Yes	No
Q3	Yes	Yes	No	No	No	No	No	No	Yes	Yes
Q4	Yes	Yes	No	No	No	No	No	No	Yes	No
Q5	Yes	Yes	No	Yes	No	No	No	No	Yes	Yes
Q6	Manually	Manually	Design patterns	On/off or abnormal values	None	None	None	None	None	Manually or recorded from sensor
Q7	All-transition-coverage criterion	All scenarios executed	Code coverage	Top-down or botton-up GUI hierarchy transverse	No more recorded events left	No more recorded events left	No more recorded events left	All edges	Breadth first search	No more recorded events left
Q8	Yes	No	Yes	Yes	No	No	No	Yes	Yes	No
Q9	Black-box	Black-box	White-box	Black-box	Black-box	Black-box	Black-box	Black-box	Black-box	Black-box and white-box
Q10	No	No	No	No	No	No	No	No	None	No
Q11	Automatic	Semi-automatic	Automatic	Automatic	Semi-automatic	Semi-automatic	Semi-automatic	Semi-automatic	Automatic	Semi-automatic

Conclusions and future works

In this work, a systematic mapping was carried out in order to identify and investigate tools that allow the automation of testing Android context-aware applications. A total of 6648 studies were obtained, 68 of which were considered as relevant publications when taking into account our research questions. These works were first analyzed according to the conference publication, year, country, and authors. The main result of this first analysis was the identification of research groups in the area of interest.

Another important contribution of this work was the identification of 80 Android application testing tools. From these tools, we identified which techniques they implement, which generate test cases, which execute test cases, which are the test methods, which ones are available for download, and which ones are most commonly used. We noticed that 40% of Android testing tools implement

GUI testing. We also note that, among the tools that generate test cases, 42% use MBT as the generation strategy approach.

The main contribution and objective of this work was the identification of context-aware Android application testing tools and the analysis of their limitations. We have identified 10 tools that support the test of context-aware applications. Five of these tools have been developed explicitly for context-aware applications, and five have been developed for general applications, but they also support context-aware features testing.

In our work, we have not done any experimental studies to evaluate other features of the tools such as test case generation time and number of revealed failures. This would require access to all tools, but 3 of the 10 tools are not available for download. Therefore, as future work, we will continue trying to gain access to the tools we could not download to test the same applications using the 10 found tools. This execution may reveal behaviors or characteristics that the authors of the tools have not expected, as well as giving more richness to the evaluation conducted in our work. We will also investigate techniques for generating test cases for context-aware applications that use high-level context with support for asynchronous context variation. From the identified techniques, we will define models and algorithms that allow automatic generation of test cases with asynchronous context variation, possibly using scenarios that are more likely to fail.

Abbreviations

APIs: Application programming interfaces; ART: Android Runtime; CASS: Context-aware software systems; GPS: Geographical Positioning System; GSM: Global System for Mobile Communications; GUI: Graphical user interfaces; HAL: Hardware abstraction layer; ICSE: International Conference on Software Engineering; JVM: Java virtual machine; MBT: Model-based testing; OS: Operating system; RQ: Research question; SMS: Systematic mapping study; TADS: Testing Application to Device State; UI: User interface; UML: Unified Modeling Language

Acknowledgements

Not applicable.

Authors' contributions

The SMS was planned by DR, WA, and PM. The study searching, reading, and data analysis was performed by DR, supervised by WA and PM. All authors reviewed and approved the final manuscript.

Authors' information

M. Diego Rodrigues is professor at the Federal Institute of Pernambuco (IFPE). Ph.D. Patrícia Machado is professor at the Federal University of Campina Grande (UFCG). D. Wilkerson Andrade is professor at the Federal University of Campina Grande (UFCG).

Funding

This work was supported by the National Council for Scientific and Technological Development (CNPq)/Brazil (Process 437029/2018-2). PM was supported by CNPq/Brazil (Process 311239/2017-0). WA was supported by CNPq/Brazil (Process 315057/2018-1).

Availability of data and materials

All data generated by our research is available through the following links:

- http://bit.ly/ArticleArchives
- http://bit.ly/Found_Tools

Competing interests

The authors declare that they have no competing interests.

Received: 3 January 2019 Accepted: 4 November 2019
Published online: 01 December 2019

References

1. Number of Mobile Phone Users Worldwide from 2015 to 2020 (in Billions). https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users%-worldwide/. Accessed 08 Aug 2019
2. Hu C, Neamtiu I (2011) Automating gui testing for android applications. In: Proceedings of the 6th International Workshop on Automation of Software Test – AST '11. ACM, New York. pp 77–83. https://doi.org/10.1145/1982595.1982612
3. Maji AK, Hao K, Sultana S, Bagchi S (2010) Characterizing failures in mobile oses: a case study with android and symbian. In: Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering – ISSRE '10. IEEE Computer Society, Washington, DC. pp 249–258. https://doi.org/10.1109/ISSRE.2010.45
4. Muccini H, Di Francesco A, Esposito P (2012) Software testing of mobile applications: challenges and future research directions. In: Proceedings of the 7th International Workshop on Automation of Software Test – AST '12. IEEE Press, Piscataway. pp 29–35. http://dl.acm.org/citation.cfm?id=2663608.2663615
5. Wasserman AI (2010) Software engineering issues for mobile application development. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research – FoSER '10. ACM, New York. pp 397–400. https://doi.org/10.1145/1882362.1882443
6. Choudhary SR, Gorla A, Orso A (2015) Automated test input generation for android: are we there yet? (e). In: Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) – ASE '15. IEEE Computer Society, Washington, DC. pp 429–440. https://doi.org/10.1109/ASE.2015.89
7. Mobile Operating System Market Share Worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed 08 Aug 2019
8. Dey AK (2001) Understanding and using context. Personal Ubiquitous Comput 5(1):4–7. https://doi.org/10.1007/s007790170019
9. Abowd GD, Dey AK, Brown PJ, Davies N, Smith M, Steggles P (1999) Towards a better understanding of context and context-awareness. In: Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing – HUC '99. Springer, London. pp 304–307. http://dl.acm.org/citation.cfm?id=647985.743843
10. Android Platform Architecture. https://developer.android.com/guide/platform/index.html. Accessed 09 Aug 2019
11. Google I/O. https://events.google.com/io. Accessed 10 Aug 2019
12. Kechagia M, Mitropoulos D, Spinellis D (2015) Charting the API minefield using software telemetry data. Empirical Softw Eng 20(6):1785–1830. https://doi.org/10.1007/s10664-014-9343-7
13. Gross F, Fraser G, Zeller A (2012) EXSYST: Search-based GUI testing. In: 2012 34th International Conference on Software Engineering (ICSE). pp 1423–1426. https://doi.org/10.1109/ICSE.2012.6227232
14. Mariani L, Pezze M, Riganelli O, Santoro M (2012) Autoblacktest: Automatic black-box testing of interactive applications. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp 81–90. https://doi.org/10.1109/ICST.2012.88
15. Memon A, Banerjee I, Nagarajan A (2003) GUI ripping: Reverse engineering of graphical user interfaces for testing. In: Proceedings of the 10th Working Conference on Reverse Engineering – WCRE '03. IEEE Computer Society, Washington, DC. p 260. http://dl.acm.org/citation.cfm?id=950792.951350
16. Yue S, Yue S, Smith R (2016) A survey of testing context-aware software: challenges and resolution. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP) 2016. IEEE Comput Soc, Las Vegas. pp 102–108
17. Wang Z, Elbaum S, Rosenblum DS (2007) Automated generation of context-aware tests. In: Proceedings of the 29th International Conference on Software Engineering – ICSE '07. IEEE Computer Society, Washington, DC. pp 406–415. https://doi.org/10.1109/ICSE.2007.18

18. Moran K, Linares-Vasquez M, Bernal-Cardenas C, Vendome C, Poshyvanyk D (2017) Crashscope: A practical tool for automated testing of android applications. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp 15–18. https://doi.org/10.1109/ICSE-C.2017.16
19. Li Y, Yang Z, Guo Y, Chen X (2017) Droidbot: A lightweight UI-guided test input generator for android. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp 23–26. https://doi.org/10.1109/ICSE-C.2017.8
20. Brown PJ (1996) The stick-e document: a framework for creating context-aware applications. In: Proceedings of EP'96, Palo Alto. pp 182–196. http://www.cs.kent.ac.uk/pubs/1996/396. 10 Aug 2019
21. Franklin D., Flachsbart J. (1998) All gadget and no representation makes jack a dull environment. In: AAAI 1998 Spring Symposium on Intelligent Environments. pp 155–160. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.1668
22. Ward A, Jones A, Hopper A (1997) A new location technique for the active office. IEEE Pers Commun 4(5):42–47. https://doi.org/10.1109/98.626982
23. Rodden T, Chervest K, Davies N, Dix A (1998) Exploiting context in HCI design for mobile systems. In: Workshop on Human Computer Interaction with Mobile Devices. pp 21–22. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.1279. https://www.alandix.com/academic/papers/exploting-context-1998/
24. Hull R, Neaves P, Bedford-Roberts J (1997) Towards situated computing. In: Digest of Papers. First International Symposium on Wearable Computers. pp 146–153. https://doi.org/10.1109/ISWC.1997.629931
25. Abowd GD, Dey AK, Brown PJ, Davies N, Smith M, Steggles P (1999) Towards a better understanding of context and context-awareness. In: Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing – HUC '99. Springer, London. pp 304–307. http://dl.acm.org/citation.cfm?id=647985.743843. 10 Aug 2019
26. Bettini C, Brdiczka O, Henricksen K, Indulska J, Nicklas D, Ranganathan A, Riboni D (2010) A survey of context modelling and reasoning techniques. Pervasive Mob Comput 6(2):161–180. https://doi.org/10.1016/j.pmcj.2009.06.002
27. Matalonga S, Rodrigues F, Travassos GH (2017) Characterizing testing methods for context-aware software systems: results from a quasi-systematic literature review. J Syst Softw 131:1–21. https://doi.org/10.1016/j.jss.2017.05.048
28. Guinea AS, Nain G, Traon YL (2016) A systematic review on the engineering of software for ubiquitous systems. J Syst Softw 118:251–276. https://doi.org/10.1016/j.jss.2016.05.024
29. Matalonga S, Rodrigues F, Travassos G (2015) Challenges in testing context aware software systems. In: Brazilian Conference on Software: Theory and Practice – CBSOFT '15. pp 51–60. https://doi.org/10.13140/RG.2.1.3361.6080
30. Sanders J, Walcott KR (2018) Tads: Automating device state to android test suite testing. In: Proceedings of the 2018 International Conference on Wireless Networks – ICWN'18. pp 10–14. https://csce.ucmss.com/cr/books/2018/LFS/CSREA2018/ICW4285.pdf. https://csce.ucmss.com/cr/books/2018/ConferenceReport?ConferenceKey=ICW
31. Holzmann C, Steiner D, Riegler A, Grossauer C (2017) An android toolkit for supporting field studies on mobile devices. In: Proceedings of the 16th International Conference on Mobile and Ubiquitous Multimedia – MUM '17. ACM, New York. pp 473–479. https://doi.org/10.1145/3152832.3157814
32. Usman A, Ibrahim N, Salihu IA (2018) Comparative study of mobile applications testing techniques for context events. Adv Sci Lett 24(10)
33. Petersen K, Feldt R, Mujtaba S, Mattsson M (2008) Systematic mapping studies in software engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering–EASE'08. BCS Learning & Development Ltd., Swindon. pp 68–77. http://dl.acm.org/citation.cfm?id=2227115.2227123
34. Kitchenham B, Charters S (2007) Guidelines for performing Systematic Literature Reviews in Software Engineering. http://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf. https://www.bibsonomy.org/bibtex/227b256010a48688388374cf83b619b54/msn
35. Start Tool Home Page. http://lapes.dc.ufscar.br/tools/start_tool. Accessed 10 Aug 2019
36. Imparato G (2015) A combined technique of GUI ripping and input perturbation testing for android apps. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2 – ICSE '15. IEEE Press, Piscataway. pp 760–762. http://dl.acm.org/citation.cfm?id=2819009.2819159
37. Vieira V, Holl K, Hassel M (2015) A context simulator as testing support for mobile apps. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing – SAC '15. ACM, New York. pp 535–541. https://doi.org/10.1145/2695664.2695782
38. Li A, Qin Z, Chen M, Liu J (2014) Adautomation: An activity diagram based automated gui testing framework for smartphone applications. In: Proceedings of the 2014 Eighth International Conference on Software Security and Reliability – SERE '14. IEEE Computer Society, Washington, DC. pp 68–77. https://doi.org/10.1109/SERE.2014.20
39. Yang W, Prasad MR, Xie T (2013) A grey-box approach for automated GUI-model generation of mobile applications. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering–FASE'13. Springer, Berlin, Heidelberg. pp 250–265. https://doi.org/10.1007/978-3-642-37057-1_19
40. Amalfitano D, Amatucci N, Fasolino AR, Tramontana P (2015) Agrippin: A novel search based testing technique for android applications. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile–DeMobile 2015. ACM, New York. pp 5–12. https://doi.org/10.1145/2804345.2804348
41. Amalfitano D, Fasolino AR, Tramontana P, Ta BD, Memon AM (2015) Mobiguitar: Automated model-based testing of mobile apps. IEEE Software 32(5):53–59. https://doi.org/10.1109/MS.2014.55
42. Amalfitano D, Fasolino AR, Tramontana P, Amatucci N (2013) Considering context events in event-based testing of mobile applications. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. pp 126–133. https://doi.org/10.1109/ICSTW.2013.22
43. Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM (2012) Using gui ripping for automated testing of android applications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering–ASE 2012. ACM, New York. pp 258–261. https://doi.org/10.1145/2351676.2351717
44. Griebe T, Gruhn V (2014) A model-based approach to test automation for context-aware mobile applications. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing–SAC '14. ACM, New York. pp 420–427. https://doi.org/10.1145/2554850.2554942
45. Silva DB, Endo AT, Eler MM, Durelli VHS (2016) An analysis of automated tests for mobile android applications. In: 2016 XLII Latin American Computing Conference (CLEI). pp 1–9. https://doi.org/10.1109/CLEI.2016.7833334
46. Prathibhan CM, Malini A, Venkatesh N, Sundarakantham K (2014) An automated testing framework for testing android mobile applications in the cloud. In: 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies. pp 1216–1219. https://doi.org/10.1109/ICACCCT.2014.7019292
47. Anand S, Naik M, Harrold MJ, Yang H (2012) Automated concolic testing of smartphone apps. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering–FSE '12. ACM, New York. pp 59–15911. https://doi.org/10.1145/2393596.2393666
48. Zaeem RN, Prasad MR, Khurshid S (2014) Automated generation of oracles for testing user-interaction features of mobile apps. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp 183–192. https://doi.org/10.1109/ICST.2014.31
49. Villanes IK, Costa EAB, Dias-Neto AC (2015) Automated mobile testing as a service (AM-TaaS). In: 2015 IEEE World Congress on Services. pp 79–86. https://doi.org/10.1109/SERVICES.2015.20
50. Coppola R, Raffero E, Torchiano M (2016) Automated mobile ui test fragility: an exploratory assessment study on android. In: Proceedings of the 2Nd International Workshop on User Interface Test Automation–INTUITEST 2016. ACM, New York. pp 11–20. https://doi.org/10.1145/2945404.2945406
51. Jensen CS, Prasad MR, Møller A (2013) Automated testing with targeted event sequence generation. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis–ISSTA 2013. ACM, New York, NY, USA. pp 67–77. https://doi.org/10.1145/2483760.2483777
52. Moran K, Linares-Vásquez M, Bernal-Cárdenas C, Vendome C, Poshyvanyk D (2016) Automatically discovering, reporting and

reproducing android application crashes. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp 33–44. https://doi.org/10.1109/ICST.2016.34
53. Haoyin LV (2017) Automatic android application GUI testing - a random walk approach. In: 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET). pp 72–76. https://doi.org/10.1109/WiSPNET.2017.8299722
54. Wang P, Liang B, You W, Li J, Shi W (2014) Automatic android GUI traversal with high coverage. In: 2014 Fourth International Conference on Communication Systems and Network Technologies. pp 1161–1166. https://doi.org/10.1109/CSNT.2014.236
55. Anbunathan R, Basu A (2017) Automation framework for test script generation for android mobile. In: 2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT). pp 1914–1918. https://doi.org/10.1109/RTEICT.2017.8256930
56. Liu CH, Lu CY, Cheng SJ, Chang KY, Hsiao YC, Chu WM (2014) Capture-replay testing for android applications. In: 2014 International Symposium on Computer, Consumer and Control. pp 1129–1132. https://doi.org/10.1109/IS3C.2014.293
57. McAfee P, Mkaouer MW, Krutz DE (2017) Cate: Concolic android testing using java pathfinder for android applications. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). pp 213–214. https://doi.org/10.1109/MOBILESoft.2017.35
58. Nguyen CD, Marchetto A, Tonella P (2012) Combining model-based and combinatorial testing for effective test case generation. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis–ISSTA 2012. ACM, New York. pp 100–110. https://doi.org/10.1145/2338965.2336765
59. Anbunathan R, Basu A (2015) Data driven architecture based automated test generation for android mobile. In: 2015 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC). pp 1–5. https://doi.org/10.1109/ICCIC.2015.7435772
60. Ye H, Cheng S, Zhang L, Jiang F (2013) Droidfuzzer: Fuzzing the android apps with intent-filter tag. In: Proceedings of International Conference on Advances in Mobile Computing & Multimedia–MoMM '13. ACM, New York. pp 68–686874. https://doi.org/10.1145/2536853.2536881
61. Jamrozik K, Zeller A (2016) Droidmate: A robust and extensible test generator for android. In: Proceedings of the International Conference on Mobile Software Engineering and Systems–MOBILESoft '16. ACM, New York. pp 293–294. https://doi.org/10.1145/2897073.2897716
62. Machiry A, Tahiliani R, Naik M (2013) Dynodroid: An input generation system for android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering–ESEC/FSE 2013. ACM, New York. pp 224–234. https://doi.org/10.1145/2491411.2491450
63. Hu G, Yuan X, Tang Y, Yang J (2014) Efficiently, effectively detecting mobile app bugs with appdoctor. In: Proceedings of the Ninth European Conference on Computer Systems–EuroSys '14. ACM, New York. pp 18–11815. https://doi.org/10.1145/2592798.2592813
64. Song W, Qian X, Huang J (2017) Ehbdroid: Beyond GUI testing for android applications. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering–ASE 2017. IEEE Press, Piscataway. pp 27–37. http://dl.acm.org/citation.cfm?id=3155562.3155570
65. Linares-Vásquez M (2015) Enabling testing of android apps. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2–ICSE '15. IEEE Press, Piscataway. pp 763–765. http://dl.acm.org/citation.cfm?id=2819009.2819160
66. Mahmood R., Mirzaei N., Malek S. (2014) Evodroid: Segmented evolutionary testing of android apps. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering–FSE 2014. ACM, New York. pp 599–609. https://doi.org/10.1145/2635868.2635896
67. van der Merwe H, van der Merwe B, Visser W (2014) Execution and property specifications for jpf-android. SIGSOFT Softw Eng Notes 39(1):1–5. https://doi.org/10.1145/2557833.2560576
68. van der Merwe H, van der Merwe B, Visser W (2012) Verifying android applications using java pathfinder. SIGSOFT Softw Eng Notes 37(6):1–5. https://doi.org/10.1145/2382756.2382797
69. Meng Z, Jiang Y, Xu C (2015) Facilitating reusable and scalable automated testing and analysis for android apps. In: Proceedings of the 7th Asia-Pacific Symposium on Internetware–Internetware '15. ACM, New York, NY, USA. pp 166–175. https://doi.org/10.1145/2875913.2875937
70. Su T (2016) Fsmdroid: Guided GUI testing of android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion–ICSE '16. ACM, New York, NY, USA. pp 689–691. https://doi.org/10.1145/2889160.2891043
71. Hu Y, Neamtiu I (2016) Fuzzy and cross-app replay for smartphone apps. In: Proceedings of the 11th International Workshop on Automation of Software Test–AST '16. ACM, New York. pp 50–56. https://doi.org/10.1145/2896921.2896925
72. Choi W, Necula G, Sen K (2013) Guided GUI testing of android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications–OOPSLA '13. ACM, New York. pp 623–640. https://doi.org/10.1145/2509136.2509552
73. Paulovsky F., Pavese E., Garbervetsky D. (2017) High-coverage testing of navigation models in android applications. In: 2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST). pp 52–58. https://doi.org/10.1109/AST.2017.6
74. Hu Y, Azim T, Neamtiu I (2015) Improving the android development lifecycle with the VALERA record-and-replay approach. In: Proceedings of the 3rd International Workshop on Mobile Development Lifecycle–MobileDeLi 2015. ACM, New York. pp 7–8. https://doi.org/10.1145/2846661.2846670
75. Qin Z, Tang Y, Novak E, Li Q (2016) Mobiplay: A remote execution based record-and-replay tool for mobile applications. In: Proceedings of the 38th International Conference on Software Engineering–ICSE '16. ACM, New York. pp 571–582. https://doi.org/10.1145/2884781.2884854
76. Lin YD, Rojas JF, Chu ETH, Lai YC (2014) On the accuracy, efficiency, and reusability of automated test oracles for android devices. IEEE Trans Softw Eng 40(10):957–970. https://doi.org/10.1109/TSE.2014.2331982
77. Wen HL, Lin CH, Hsieh TH, Yang CZ (2015) Pats: A parallel GUI testing framework for android applications. In: 2015 IEEE 39th Annual Computer Software and Applications Conference Vol. 2. pp 210–215. https://doi.org/10.1109/COMPSAC.2015.80
78. Hao S, Liu B, Nath S, Halfond WGJ, Govindan R (2014) Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services–MobiSys '14. ACM, New York. pp 204–217. https://doi.org/10.1145/2594368.2594390
79. Lam W, Wu Z, Li D, Wang W, Zheng H, Luo H, Yan P, Deng Y, Xie T (2017) Record and replay for android: Are we there yet in industrial cases?. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering–ESEC/FSE 2017. ACM, New York. pp 854–859. https://doi.org/10.1145/3106237.3117769
80. Mirzaei N, Garcia J, Bagheri H, Sadeghi A, Malek S (2016) Reducing combinatorics in GUI testing of android applications. In: Proceedings of the 38th International Conference on Software Engineering–ICSE '16. ACM, New York. pp 559–570. https://doi.org/10.1145/2884781.2884853
81. Gomez L, Neamtiu I, Azim T, Millstein T (2013) Reran: Timing- and touch-sensitive record and replay for android. In: Proceedings of the 2013 International Conference on Software Engineering–ICSE '13. IEEE Press, Piscataway. pp 72–81. http://dl.acm.org/citation.cfm?id=2486788.2486799
82. Zun D, Qi T, Chen L (2016) Research on automated testing framework for multi-platform mobile applications. In: 2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS). pp 82–87. https://doi.org/10.1109/CCIS.2016.7790229
83. de Cleva Farto G, Endo AT (2017) Reuse of model-based tests in mobile apps. In: Proceedings of the 31st Brazilian Symposium on Software Engineering–SBES'17. ACM, New York. pp 184–193. https://doi.org/10.1145/3131151.3131160
84. Mao K, Harman M, Jia Y (2016) Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis–ISSTA 2016. ACM, New York. pp 94–105. https://doi.org/10.1145/2931037.2931054
85. Neto NML, Vilain P, Mello RdS (2016) Segen: Generation of test cases for selenium and selendroid. In: Proceedings of the 18th International Conference on Information Integration and Web-based Applications

and Services–iiWAS '16. ACM, New York. pp 433–442. https://doi.org/10.1145/3011141.3011154
86. Mirzaei N, Bagheri H, Mahmood R, Malek S (2015) Sig-droid: Automated system input generation for android applications. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). pp 461–471. https://doi.org/10.1109/ISSRE.2015.7381839
87. Adamsen CQ, Mezzetti G, Møller A (2015) Systematic execution of android test suites in adverse conditions. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis–ISSTA 2015. ACM, New York. pp 83–93. https://doi.org/10.1145/2771783.2771786
88. Salihu I. A., Ibrahim R. (2016) Systematic exploration of android apps' events for automated testing. In: Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media–MoMM '16. ACM, New York. pp 50–54. https://doi.org/10.1145/3007120.3011072
89. Azim T, Neamtiu I (2013) 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. pp 641–660. https://doi.org/10.1145/2509136.2509549
90. Kaasila J, Ferreira D, Kostakos V, Ojala T (2012) Testdroid: Automated remote UI testing on android. In: Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia–MUM '12. ACM, New York. pp 28–1284. https://doi.org/10.1145/2406367.2406402
91. Morgado IC, Paiva ACR (2015) The impact tool: Testing ui patterns on mobile applications. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp 876–881. https://doi.org/10.1109/ASE.2015.96
92. Zhauniarovich Y, Philippov A, Gadyatskaya O, Crispo B, Massacci F (2015) Towards black box testing of android apps. In: 2015 10th International Conference on Availability, Reliability and Security. pp 501–510. https://doi.org/10.1109/ARES.2015.70
93. Li X, Jiang Y, Liu Y, Xu C, Ma X, Lu J (2014) User guided automation for testing mobile apps. In: 2014 21st Asia-Pacific Software Engineering Conference Vol. 1. pp 27–34. https://doi.org/10.1109/APSEC.2014.13
94. Hu Y, Neamtiu I (2016) Valera: An effective and efficient record-and-replay tool for android. In: Proceedings of the International Conference on Mobile Software Engineering and Systems–MOBILESoft '16. ACM, New York. pp 285–286. https://doi.org/10.1145/2897073.2897712
95. Liu Y, Xu C (2013) Veridroid: Automating android application verification. In: Proceedings of the 2013 Middleware Doctoral Symposium–MDS '13. ACM, New York. pp 5–156. https://doi.org/10.1145/2541534.2541594
96. Hu Y, Azim T, Neamtiu I (2015) Versatile yet lightweight record-and-replay for android. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications–OOPSLA 2015. ACM, New York. pp 349–366. https://doi.org/10.1145/2814270.2814320
97. Cao C, Meng C, Ge H, Yu P, Ma X (2017) Xdroid: Testing android apps with dependency injection. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC) Vol. 1. pp 214–223. https://doi.org/10.1109/COMPSAC.2017.268
98. Ami AS, Hasan MM, Rahman MR, Sakib K (2018) Mobicomonkey: Context testing of android apps. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems–MOBILESoft '18. ACM, New York. pp 76–79. https://doi.org/10.1145/3197231.3197234
99. Yan J, Pan L, Li Y, Yan J, Zhang J (2018) Land: A user-friendly and customizable test generation tool for android apps. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis–ISSTA 2018. ACM, New York. pp 360–363. https://doi.org/10.1145/3213846.3229500
100. Chen J, Han G, Guo S, Diao W (2018) Fragdroid: Automated user interface interaction with activity and fragment analysis in android applications. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp 398–409. https://doi.org/10.1109/DSN.2018.00049
101. Koroglu Y, Sen A (2018) Tcm: Test case mutation to improve crash detection in android. In: Russo A, Schürr A (eds). Fundamental Approaches to Software Engineering. Springer, Cham. pp 264–280
102. Google Scholar. https://scholar.google.com. Accessed 10 Aug 2019
103. Azim T, Neamtiu I (2013) Targeted and depth-first exploration for systematic testing of android apps. SIGPLAN Not 48(10):641–660. https://doi.org/10.1145/2544173.2509549
104. Zeng X, Li D, Zheng W, Xia F, Deng Y, Lam W, Yang W, Xie T (2016) Automated test input generation for android: are we really there yet in an industrial case?. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering–FSE 2016. ACM, New York. pp 987–992. https://doi.org/10.1145/2950290.2983958
105. Amalfitano D, Fasolino AR, Tramontana P, Ta BD, Memon AM (2015) MobiGUITAR: Automated Model-Based Testing of Mobile Apps. IEEE Softw 32(5):53-59. https://doi.org/10.1109/MS.2014.55.
106. Linares-Vásquez M, Bernal-Cardenas C, Moran K, Poshyvanyk D (2017) How do developers test android applications?. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp 613–622. https://doi.org/10.1109/ICSME.2017.47
107. Villanes IK, Ascate SM, Gomes J, Dias-Neto AC (2017) What are software engineers asking about android testing on stack overflow?. In: Proceedings of the 31st Brazilian Symposium on Software Engineering–SBES'17. ACM, New York. pp 104–113. https://doi.org/10.1145/3131151.3131157
108. Helppi V-V (2016) Calabash 101 - Basics, Getting Started, and Advanced Tips. https://offers.bitbar.com/ebook/calabash-101-basics-getting-started-and-advanced-tips. Access 14 Aug 2019
109. Automated UI Testing with Cucumber and Calabash. https://praeclarum.org/2014/01/17/automated-ui-testing-with-cucumber-and-calabash.html. Accessed 14 Aug 2019
110. Mirza AM, Khan MNA (2018) An automated functional testing framework for context-aware applications. IEEE Access 6:46568–46583. https://doi.org/10.1109/ACCESS.2018.2865213
111. Luo C, Kuutila M, Klakegg S, Ferreira D, Flores H, Goncalves J, Mäntylä M, Kostakos V (2017) Testaware: A laboratory-oriented testing tool for mobile context-aware applications. Proc ACM Interact Mob Wearable Ubiquitous Technol 1(3):80–18029. https://doi.org/10.1145/3130945
112. Griebe T, Hesenius M, Gruhn V (2015) Towards automated UI-tests for sensor-based mobile applications. In: Intelligent Software Methodologies, Tools and Techniques - 14th International Conference, SoMeT 2015, Naples, Italy, September 15-17, 2015. Proceedings. pp 3–17. https://doi.org/10.1007/978-3-319-22689-7_1
113. Shrestha A, Biel B, Griebe T, Gruhn V (2011) A framework for building and operating context-aware mobile applications. In: Mobile Wireless Middleware, Operating Systems, and Applications - 4th International ICST Conference, Mobilware 2011, London, UK, June 22-24, 2011, Revised Selected Papers. pp 135–142. https://doi.org/10.1007/978-3-642-30607-5_13
114. Samsung Health. https://health.apps.samsung.com. Accessed 10 Aug 2019

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

Submit your next manuscript at ▶ springeropen.com