TestAWARE: A Laboratory-Oriented Testing Tool for Mobile Context-Aware Applications

CHU LUO, The University of Melbourne
MIIKKA KUUTILA, SIMON KLAKEGG and DENZIL FERREIRA, University of Oulu
HUBER FLORES, University of Helsinki
JORGE GONCALVES, The University of Melbourne
MIKA MÄNTYLÄ, University of Oulu
VASSILIS KOSTAKOS, The University of Melbourne

Although mobile context instrumentation frameworks have simplified the development of mobile context-aware applications, it remains challenging to test such applications. In this paper, we present TestAWARE that enables developers to systematically test context-aware applications in laboratory settings. To achieve this, TestAWARE is able to download, replay and emulate contextual data on either physical devices or emulators. To support both white-box and black-box testing, TestAWARE has been implemented as a novel structure with a mobile client and code library. In black-box testing scenarios, developers can manage data replay through the mobile client, without writing testing scripts or modifying the source code of the targeted application. In white-box testing scenarios, developers can manage data replay and test functional/non-functional properties of the targeted application by writing testing scripts using the code library. We evaluated TestAWARE by quantifying its maximal data replay speed, and by conducting a user study with 13 developers. We show that TestAWARE can overcome data synchronisation challenges, and found that PC-based emulators can replay data significantly faster than physical smartphones and tablets. The user study highlights the usefulness of TestAWARE in the systematic testing of mobile context-aware applications in laboratory settings.

CCS Concepts: • **Human-centered computing → Ubiquitous and mobile computing**; • **Software and its engineering → Software verification and validation**

Additional Key Words and Phrases: Sensors, context aware computing, machine learning, mobile interaction, mobile sensing

ACM Reference format:

Chu Luo, Miikka Kuutila, Simon Klakegg, Denzil Ferreira, Huber Flores, Jorge Goncalves, Mika Mäntylä, and Vassilis Kostakos. 2017. TestAWARE: A Laboratory-Oriented Testing Tool for Mobile Context-Aware Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 3, Article 80 (September 2017), 29 pages.
DOI: http://doi.org/10.1145/3130945

Authors' addresses: C. Luo, J. Goncalves, V. Kostakos, University of Melbourne, VIC-3010 Parkville, Australia, email: CHUL3@student.unimelb.edu.au, {firstname.lastname}@unimelb.edu.au; M. Kuutila, S. Klakegg, D. Ferreira, M. Mäntylä, University of Oulu, Oulu 90014, Finland, email: {firstname.lastname}@oulu.fi; H. Flores, University of Helsinki, Helsinki 00014, Finland, email: huber.flores@helsinki.fi.

2474-9567/2017/9-ART80 $15.00

DOI: http://doi.org/10.1145/3130945

1 INTRODUCTION

Mobile devices such as smartphones and tablets now embed a variety of sensors (e.g., accelerometer, gyroscope, GPS and magnetometer) that can provide context-aware services to increasingly large user bases. Driven by the richness of heterogeneous data and machine learning algorithms, context-aware applications are becoming increasingly popular across a variety of domains, including quantified self [27], urban sensing and e-health. To simplify the development of mobile context-aware applications, researchers have built a significant number of context management frameworks such as AWARE [10], EmotionSense [23] and Sensor Data Collection Framework (SDCF) [4]. Although development has become easier, testing of such context-aware applications remains challenging. For instance, it is challenging to test such applications in laboratory settings, and therefore developers and researchers have to resort to expensive pilots with users [21,22].

There are several reasons why testing mobile context-aware applications remains challenging. First, due to the heterogeneity of contextual data and complexity of algorithms, static testing by reading the source code is impractical for developers to conduct efficiently. Second, it is time-consuming and expensive to robustly test mobile context-aware applications in realistic pilot studies, due to the potentially large number of context states that require testing (e.g., specific time, locations and user groups). Thirdly, certain types of contextual data, such as human falls or applications crashes, are too rare for applications to capture during testing [17].

To alleviate these problems, researchers have proposed several testing techniques and tools. For example, ContextViewer [6] is a visualisation and processing tool that helps developers understand the distribution of contextual data values from different sensors. KnowMe [9] is a tool that can replay online data on a PC, and send test cases to context-aware applications for processing. Similarly, MobiPlay [22] is a remote testing tool that allows developers to test mobile applications on a server using recorded sensor data such as GPS coordinates and acceleration.

Despite these efforts, developers may still struggle with the lack of suitable datasets for testing. To overcome the lack of such datasets, researchers have designed several techniques to manipulate contextual data for testing specific applications. For instance, CRASHDROID [30] is a specialised tool for reproducing and replaying bug reports of applications for testing. Similarly, to confirm whether mobile crowdsourcing applications trigger questions in the correct context, one proposed approach is the integration of real-time, historical and simulated data to construct the intended context [17]. Similar work [24] can create, test and simulate interactions across mobile devices and ubiquitous smart infrastructures such as smart home.

However, for systematic testing of mobile context-aware applications, we still lack a laboratory testing platform that can replay/emulate a rich variety of context from different data sources, or test functional/non-functional properties (e.g., power consumption, machine learning accuracy and processing speed). For instance, using any combination of extant tools, developers cannot replay online historical data to test applications on physical devices or emulators. Even if a combination of tools supports the testing of an application, developers have to conduct multiple rounds of testing using each tool. Most importantly, none of existing tools provide the support of white-box testing. Without such support, developers can hardly locate and fix bugs in the source code. To address this issue, we have developed TestAWARE, a laboratory-oriented tool for testing mobile context-aware applications. The tool can download, replay and construct contextual data from different sources on either physical devices or device emulators. Based on a novel architecture consisting of a mobile client and code library, it supports both black-box and white-box testing, and testing of functional/non-functional properties. Testers can conduct different types of testing, depending on their actual requirements and resources. In summary, our work makes the following contributions:

1. Conceptually, we identify a gap between existing testing tools and the requirements for laboratory-based testing of mobile context-aware applications. Existing tools raise a significant number of challenges which impede effective and efficient testing of mobile context-aware applications in laboratory settings.
2. To overcome the identified challenges, we propose TestAWARE, a laboratory-oriented testing tool that supports the testing of mobile context-aware applications by constructing the intended context and examining functional/non-functional properties. Unlike prior work supporting only one type of

testing, TestAWARE aims at a wide variety of mobile context-aware applications and testing scenarios, by incorporating heterogeneous data (i.e., sensory data, events and audio), multiple data sources (i.e., online, local and manipulated data), black-box/white-box testing, functional/non-functional property examination and the environments of device/emulator.

3. We quantify the maximal replay speed of sensory, event and audio data for testing scenarios where testers want to efficiently conduct a testing task with longitudinal datasets (e.g., maximising the replay speed for sensory data collected across multiple months or years). We show that our tool can take advantage of PC-based emulators to replay data significantly faster than actual smartphones and tablets.
4. We evaluate our tool in a user study with 13 professional mobile application developers. The results highlight the strengths of our tool for testing mobile context-aware applications in laboratory settings.

2 RELATED WORK

The development of context-aware computing systems was one of the key challenges of Ubiquitous Computing [25]. At present, there exists a plethora of context management frameworks to develop mobile context-aware applications, e.g., AWARE [10], EmotionSense [23] and SDCF [4]. However, we observe that testing techniques and tools for context-aware systems have lagged behind the state-of-the-art in context-aware computing technology [16]. Muccini et al. [21] identified several challenges of mobile context-awareness testing, e.g., rich data sources, lack of testing tools and hardware difference. Testing tools need data to reconstruct the context where context-aware applications operate. Just as understanding user intent without explicit user input is challenging [8], it remains difficult and expensive to test mobile context-aware applications without relevant contextual data and data management tools [17]. In this section, we first summarise relevant work that addresses these problems. Then we identify the gap between the state of arts and the need in testing mobile context-aware application.

2.1 Testing Mobile Context-Aware Applications

Typically, the manufacturers of mobile operating systems provide basic software development kits (SDK) to support testing. For example, Android has a testing tool called Monkey [3] which can generate user interface (UI) events and certain system-level events. Monkey is able to record emerging errors in the testing. Similarly, Android also offers an automated testing tool called Monkeyrunner [20] which allows developers to test applications without modification of the application source code.

Furthermore, developers and researchers have developed tools for testing more complex behaviours. For instance, the GUI crawler [1] can identify bugs by automatically simulating Android application executions. Similarly, Mosaic [13] is an Android-based record-and-replay tool to capture and replay user interactions across interfaces. Based on the AWARE [10] context-aware middleware, ContextViewer [6] is a visualisation and initial processing tool to help developers understand contextual datasets collected by AWARE. However, these testing tools, including industrial tools (e.g., Testdroid [14]), do not have the record-and-replay feature for contextual data.

Hence, a large body of work focuses on the features of creating or replaying context events. Amalfitano et al. [2] highlight that mobile applications can be tested by event-based techniques with the consideration of both context and UI events. Tonjes et al. [28] present a semi-automated method for test case generation and test case execution. Specific values of contextual data can be automatically created for each test case. RERAN [11] is an Android-based record-and-replay tool to collect low-level data streams such as UI events and hardware sensor data. Although it can record data from various Android sensors such as accelerometer, it is unable to capture GPS information because GPS is a stand-alone service in Android. KnowMe [9] is a tool that can replay data collected by AWARE. It allows testers to specify the speed of replay. However, it only works on PCs and can only fetch data from online databases of AWARE. MobiPlay [22] is a remote testing tool that allows developers to test Android applications using recorded datasets such as GPS and accelerometer. MobiPlay runs these applications on a remote server. The MobiPlay client on mobile devices acts as the GUI of the targeted applications. A limitation of MobiPlay is that developers cannot use datasets of other middleware from online sources or device storage. Also, developers cannot launch tests without a specialised server.

If the intended context is uncommon for a mobile device to capture, generic testing or record-and-replay tools cannot provide effective support in testing. Hence, researchers have built a number of specialised tools. Griebe and Gruhn [12] present a model-based testing method that generates an appropriately selected group of test cases according to system design models. CRASHDROID [30] is a specialised tool to reproduce and replay bug reports for testing. Also, Roalter et al. [24] propose a development tool to create, test and simulate interactions between mobile devices and ubiquitous smart infrastructures such as smart home.

However, functional-testing tools and techniques which also support the testing of non-functional properties (e.g., energy consumption, machine learning accuracy and processing speed) of applications are non-existent. The feature of testing non-functional properties is crucial because mobile devices have limited computing resources. For example, it is necessary to balance power consumption and classification accuracy for mobile context-aware applications [7]. Although several specialised tools, such as the power estimation IDE (integrated development environment) [19] and FOREPOST [18], focus on performance, it is a significant burden for developers to test functionalities and each property of an application using multiple tools.

2.2 The Gap to the Need for Mobile Context-Aware Testing

Table 1. Comparison

Tool/Method	Data Type	Data Source	B/W Box Testing	Non-functional	Environment
Monkey [3]	Event	Script	B	-	D/E
Monkeyrunner [20]	Event	Script	B	-	D/E
GUI crawler [1]	Event	Script	B	-	D/E
Mosaic [13]	Event	Record	B	-	D/E
ContextViewer [6]	Sensor	Online	-	-	-
Testdroid [14]	Event	Script	B	-	D
[2]	Event	Script	B	-	D/E
[28]	Sensor, Event	Script	B	-	-
RERAN [11]	Sensor, Event	Record	B	-	D/E
KnowMe [9]	Sensor	Online	-	-	-
MobiPlay [22]	Sensor, Event	Record	B	-	-
[12]	Sensor, Event	Model	B	-	D/E
CRASHDROID [30]	Bug Report	Record	B	-	D/E
[24]	Event	Script	B	-	-
[7]	Sensor, Event	Online	-	ML, PS, PC	D/E
[19]	-	-	-	PC	-
FOREPOST [18]	Sensor, Event	Online	-	PS	D/E
TestAWARE	Sensor, Event, Audio	Online, Local, Script	B/W	ML, PS, PC	D/E

B – Black-Box Testing, W – White-Box Testing, D – Device, E – Emulator, ML – Machine Learning, PS – Processing Speed, PC – Power Consumption

Summarising the capabilities of existing tools and methods, Table 1 frames TestAWARE in relation to the state of the art using a number of criteria:

1. *Data type*: mobile context-aware applications may collect and process various kinds of data, including hardware sensory data, software data, human input, audio and video [10]. We found that half of previous work only focused on a single data type. Also, replaying audio and video data during testing is rare in the literature. An ideal testing tool should support a wide range of data types to meet requirements of different applications.
2. *Data source*: test cases are based on testing data. For simple low-dimensional context, developers can easily generate and record ad-hoc data. For example, the battery charging status of a smartphone can be changed by connecting or disconnecting the USB cable on a computer. However, for rare or multi-dimensional context, developers may not have the opportunity to conduct systematic and exhaustive testing across the intended context. In these cases, developers can resort to re-using historical or manipulating data by writing a script. However, all existing tools only support one such way of obtaining testing data. This poses a crucial challenge for developers to reuse existing datasets. Even, some record-and-replay testing tools do not accept data from other sources. Therefore, a tool allowing multiple data sources can enable the testing of applications aiming at uncommon context and can significantly simplify the testing process.
3. *Black/white-box testing*: in the software testing process, developers normally use both two methods, black-box testing and white-box testing, to test an application. Many existing provide neither method, and the remainder of tools focus only on black-box testing, which can only provide functional feedback. Our search did not identify a single white-box testing tool for mobile context-aware applications, even though white-box testing is very important in software development. This is because black-box testing only verifies whether a software can achieve its functional objectives, but cannot ensure that all the possible states of software are valid. Using these tools alone, developers do not have any support to examine internal behaviours of their context-aware applications. Hence, providing both methods of testing is a necessary requirement for a testing tool.
4. *Non-functional testing*: non-functional properties play a crucial role in the practical use of applications. For mobile context-aware applications, testing tools should support 3 aspects which are closely related to user experience: machine learning performance, processing speed and power consumption [7]. Despite the existence of specialised testing tools, very few support this kind of testing.
5. *Environment*: during testing, developers may need to examine the states of applications that correctly operate on physical devices. Also, the measurement of processing speed requires the environment of physical devices. When developers do not have access to specific physical devices (e.g., having certain screen sizes or OS versions), they must rely on emulators. For this requirement, testing tools must support both device and emulator environments, rather than only executing tests on source code using a server or PC. Approximately half of existing tools and methods do not provide this feature.

Our analysis highlights the state of the art in mobile context-aware testing tools, and shows that there exist significant challenges in the testing of mobile context-aware applications, as also reported in literature [17]. We argue that the requirements for a testing tool for mobile context-aware applications are to support all the different testing scenarios from the criteria. The comparison across existing tools highlights a gap in the state of the art: none of them aim to work as a holistic tool for the complete testing process on diverse mobile context-aware applications and testing types. For instance, using any combination of existing tools, developers cannot use online historical data to test applications on physical devices or emulators. Even if a combination of tools supports the testing of an application, developers have to run multiple rounds of testing using each tool. Most importantly, prior work ignores the support of white-box testing. With existing tools, developers can hardly locate and fix bugs in the source code. Hence, our research addresses this gap in literature by developing TestAWARE to match the requirements set out in Table 1.

3 FUNCTIONALITY

Before describing the technical details of our system, we first present a high-level overview of how our system functions, and how it supports developers in testing mobile context-aware applications in laboratory settings. To simplify our description, and make it more concrete, we consider a scenario where a developer is testing a mobile application that performs real-time fall detection (i.e., the application detects whether the phone user falls down or not. It does not report a fall if the phone itself drops from the user.) on the phone. The application is programmed to send an email to a caregiver every time a fall event is detected by the phone.

The main challenge that the developer faces in testing this application is the effort that it takes to test it in realistic settings. Every time the application algorithm is tweaked or improved, new tests need to be conducted to ensure that the application works well in detecting fall events. Typically, a developer would compile a new version of their application, install it on a phone, and then conduct physical tests where they drop the phone under a variety of condition (e.g. drop from the hand, fall down with the phone in the pocket, etc.). This testing regime is also representative of the practices of researchers who develop context-aware applications.

TestAWARE provides developers the ability to fuse simulated, historical, and real-time data in their testing regime [17] (Fig. 1). These can be combined to "reconstruct" the indented context within which their application should be tested. This context can be recreated on a physical device, or a device emulator.

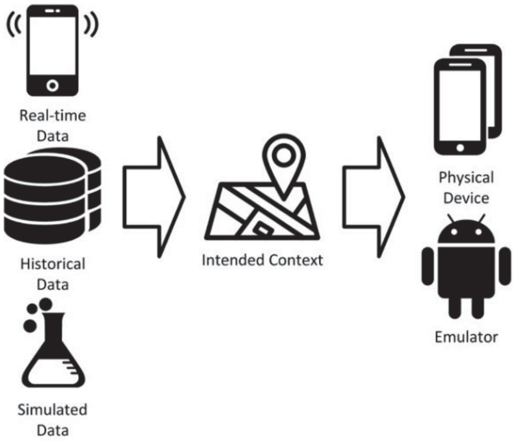


Fig. 1. TestAWARE supports the fusion [17] of real-time, historical, and simulated data during testing.

3.1 Context Data Preparation

The first step in using our tool, is that the developer needs to capture, download or generate contextual data. This data should reflect the conditions where the targeted application will be tested. In our scenario, the developer needs to record new data or download historical data from all relevant sensors, when the phone is dropped under a variety of circumstances. Therefore, the developer needs to orchestrate and record human fall events, as well as likely false-positive events (e.g., the phone dropping from a user's hand). Many kinds of context middleware, such as AWARE, can be used to record the relevant sensor data (e.g., accelerometer and gyroscope), and the data needs to be stored in an AWARE-compatible format (i.e., records with timestamps).

We note that developers can also create simulated sensor values, for example simulated accelerometer values or constant sensor values. This approach of using simulated data may be helpful in cases where rare or specific events need to be emulated. Developers can construct synthesised contextual data using the TestAWARE library. In our scenario, the developer could, for instance, create a for-loop that samples values from a normal distribution and stores those values as simulated accelerometer values using the TestAWARE library.

3.2 Black-box Testing

To conduct black-box testing of the application, the next step will be to configure the TestAWARE client. The developer should install the client and the application in the testing environment (e.g., phone or emulator).

Next, the TestAWARE client needs data details before it replays data. To achieve this, the developer should indicate where the context data is stored (either online or in local storage). According to the AWARE-compatible data format, a single file or database table represents data for a single sensor. Therefore, the developer needs to indicate the database table or local file that contains the contextual data to be replayed, and then the TestAWARE client fetches all data to make sure it is available locally for the tests.

Subsequently, the developer needs to define in the TestAWARE client a replay task: this is a definition of what sensor data should be replayed, and the timestamp range. A replay task is analogous to a set of test cases. A developer may define multiple replay tasks: some replay tasks may include more sensors that others, and different replay tasks can use different context data files. For instance, in our scenario the developer could create one replay task that uses only the accelerometer values the developer has previously recorded; another replay task can use both accelerometer and gyroscope. Additionally, the developer could create a replay task that uses accelerometer and gyroscope values captured in a lab setting, while another replay task could be defined to use the values captured from a bedroom. If an application needs a sensor to operate, but that sensor is not included in a replay task, then the device uses data from the actual physical sensor.

Finally, the developer needs to begin execution of a replay task on the TestAWARE client, and define the replay speed. Once the replay has started, the developer switches to the targeted application, and observes/monitors its behaviour. In black-box testing we would assume that the application outputs results either into a database, or perhaps via debug messages. Once the replay task is finished, the developer needs to inspect the output of the application, and identify problems or errors. In our scenario, the developer could simply inspect the timestamps when his application detected a fall event.

3.3 White-box Testing

If the developer wishes to conduct white-box testing of the application, the next step will be to import the TestAWARE library into the application. The developer then needs to edit the source code of this application to define, configure, and control the replay tasks that are desired. Effectively, the developer needs to programmatically configure the TestAWARE client for data replay. The reason for programmatic configuration is to enhance automation, and facilitate repeatable tests. Ideally, this code would appear in the application when it is ready to start processing sensor values.

There are additional changes to the source code that the developer can make, to improve testing. First, the developer can add **assertions** that the TestAWARE library provides. These assertions will be communicated, at runtime, to the TestAWARE client, and help the developer identify bugs. In our scenario, the developer would add such an assertion after a fall is detected, and indicate: a brief description of the assertion (e.g. “React to fall”), the expected value (e.g. “Message to caregiver is sent”) and the actual value (e.g., “Message to caregiver is not sent”). This assertion helps the developer test whether the application responds as expected to context. After the tests are complete, the developer can inspect all assertions that were logged during the test, to identify problems in the behaviour of the application.

Second, TestAWARE allows source code modifications to facilitate the evaluation of applications to use **machine learning assertions**. This functionality is meant to help developers verify that modifications to their machine learning code have resulted in higher accuracy during the tests. The developer needs to make sure that the replayed data has ground-truth labels. These labels are created during the Data Preparation phase, and they need to reflect the desirable behaviour of the application. For example, in our scenario the developer would include “FALL” labels with the captured accelerometer values. Then, the developer would edit the source code that processes each new incoming accelerometer values, and add an assertion using the

TestAWARE library. The assertion would compare the output of the application's machine learning classification (e.g. "FALL" or "NO-FALL") versus the ground truth label that is included in the replayed data. In case where the context recognition is not classification but a regression, the assertion compares the expected value (ground truth label) versus the predicted value (as predicted by the application).

Third, the developer can insert a code snippet while the application launches, to enable **energy consumption profiling**. This code snippet informs the TestAWARE client of the sensor energy specifications of the handset that is tested. In our scenario, the developer would create a snippet that: defines the "Nexus 5" as the handset; indicates "Accelerometer" (one of the sensor names being replayed); indicates the sensing delay (e.g. "Normal", as specified by Android documentation); and indicates the expected power consumption per hour (e.g. "0.4 mAh per hour"). The expected power consumption may be defined according to the developer's expectations, or perhaps the hardware specifications of sensors (although those tend to be inaccurate).

Finally, the developer can perform **processing speed profiling** by calling functions provided by the TestAWARE library. These functions are meant to be called before and after a time-consuming operation takes place, such as regression or classification. The developer can assign each measurement a label so that multiple modules in the code can be measured without conflict. The functions are used to measure the time that it takes for the operation to complete, and they communicate these results, in real time, to the TestAWARE client. In our scenario, the developer would add this pair of statements around the line of code that initiates classification when new accelerometer sensor values arrive. The developer would then see in the TestAWARE client the average and worst-case durations recorded during the test.

4 FUNCTIONALITY

TestAWARE consists of a mobile client and code library, as shown in Fig. 2. We will detail the functions, usage scenarios and design trade-offs of each component in the following subsections. Although TestAWARE aims at the testing only on the Android platform due to its popularity, this architecture can be deployed to build similar testing tools on other platforms, such as iOS and Windows.

As suggested by previous work [9,10], we implement both the mobile client and code library for the Android platform. The purpose of TestAWARE is to facilitate the testing of different mobile context-aware applications. Our main objective is to minimise the reliance on testing that requires the end-users of targeted applications. Conceptually, TestAWARE is able obtain and replay "context", and thus provide a reliable and repeatable setting for testing context-aware applications.

Before replaying contextual data, developers should collect relevant datasets either from online sources or from the device storage using local providers. Developers can also generate synthetic data programmatically using the data manipulator. Once contextual data is available, the developer can replay it using the data replayer, while in parallel the energy evaluator estimates the energy consumption of each sensor involved in the replay. Both the client user interface and command API (Application Programming Interface) of the code library provide replay control.

In black-box testing, developers cannot modify the targeted application. Hence, the targeted application receives and processes data, and remains ignorant of the presence of the TestAWARE client and code library. In this case, the targeted application outputs results as usual, and developers have to analyse these results through the functionality of the targeted application (e.g., monitoring the user interface, or inspecting the database of the targeted application), because the application does not send the results to the TestAWARE client.

TestAWARE allows developers to perform white-box testing by importing the code library into the targeted application. In this case, a context-aware module in the targeted application can output results to the result recorder of the TestAWARE client. Then the machine learning evaluator can generate performance analysis based on the recorded results. In addition, developers can make use of the processing speed evaluator to record the time of executions. Similar to other white-box testing methods, the limitation is that the testing requires modification in the source code of the targeted application.

TestAWARE's architecture entailed several design decisions and trade-offs. In turn, these were guided by the requirements we had set out for our tool. In Table 2 we describe how our requirements (from Table 1) were mapped to design choices in TestAWARE's architecture.

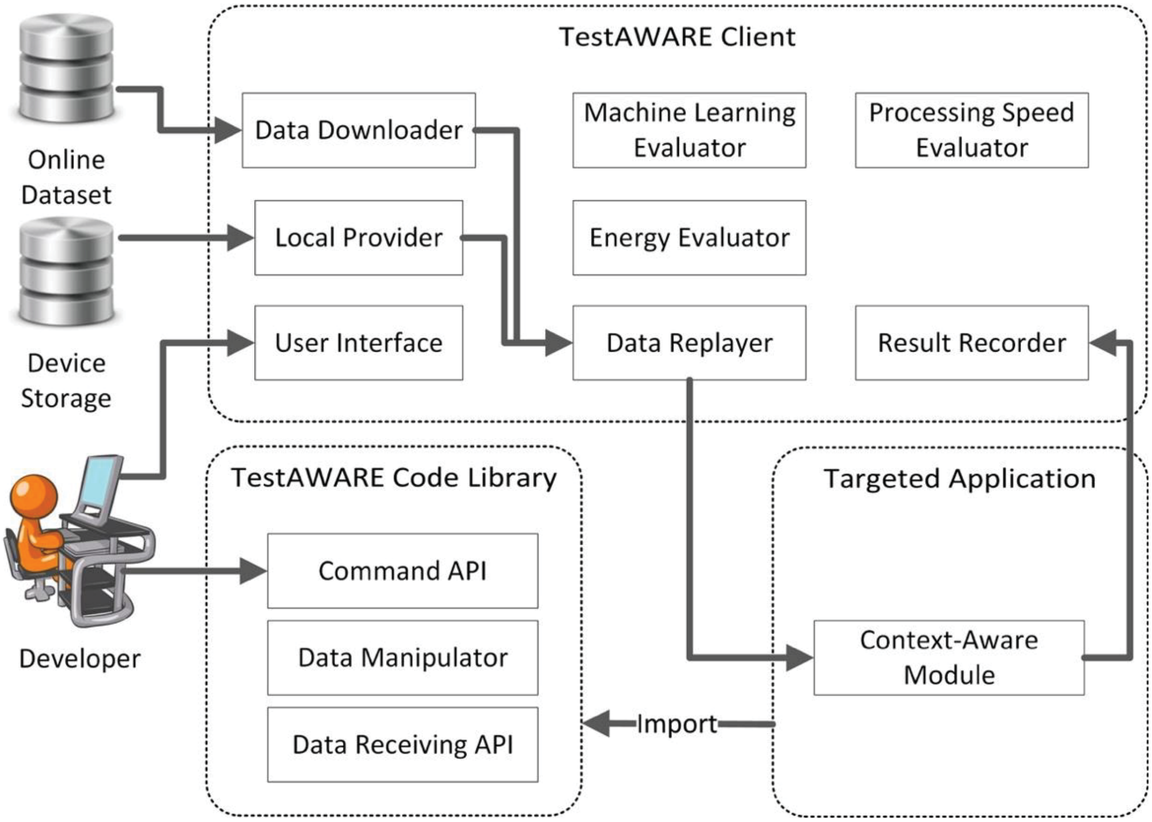


Fig. 2. TestAWARE architecture. The Client and Code Library are used to test the Targeted Application.

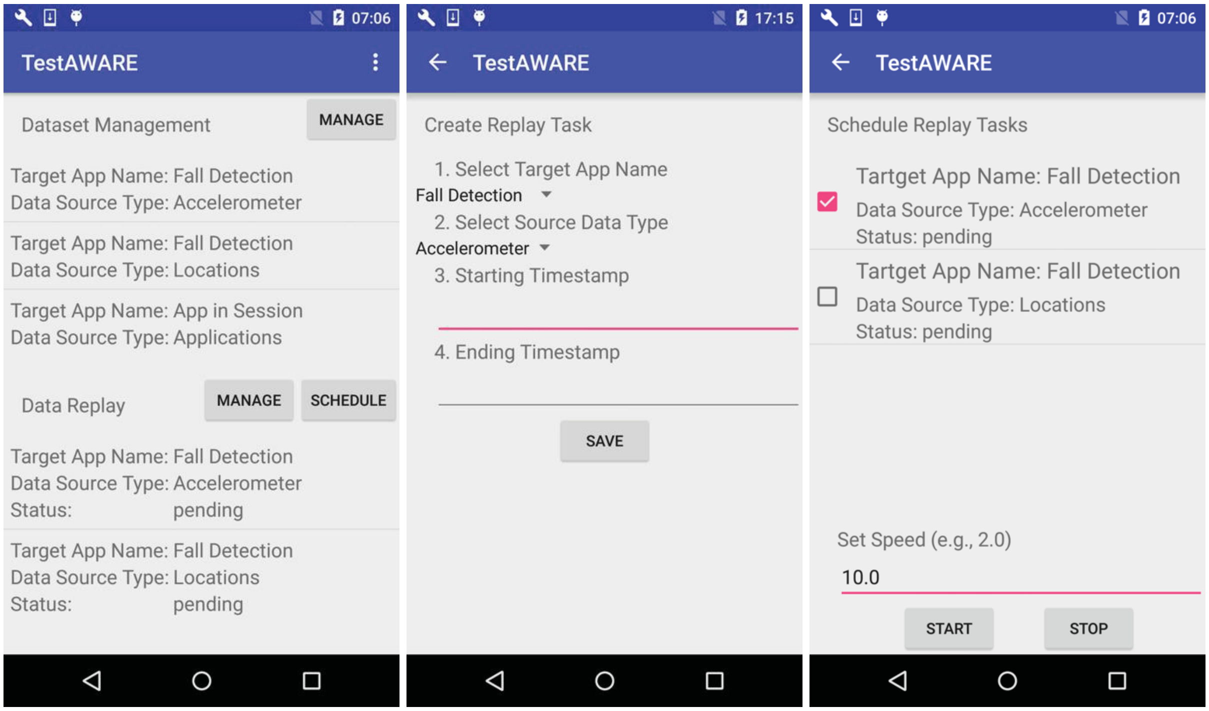
4.1 TestAWARE Client

The TestAWARE client is an Android application for mobile devices. It can also run on PC-based device emulators. During testing, the client runs simultaneously with the targeted application. The client user interface provides developers with testing data, replay data and read non-functional performance results. The client supports developers to conduct black-box testing on physical devices and emulators. Since developers cannot modify source code in the black-box testing, deploying a client running with the targeted application is the only viable approach. Fig. 3 shows screenshots of the client user interface during testing. Note that developers can also write scripts using the code library to automate the same tasks.

4.1.1 Data Downloader. The data downloader allows developers to test applications with data replayed by the device, using existing online datasets such as an AWARE database. For example, developers can use online datasets from previous projects or experiments. Specifically, developers can specify credentials for a database (including database host, username, password, data table and timestamp period). Thus, the data downloader can download the respective dataset completely or partly to the device according to the user specifications.

Table 2. Mapping from requirements to design choices.

Requirement	Design Choice
Support the replay of heterogeneous data	Incorporate support for sensory, event and audio data in data downloader, local provider, data manipulator and data replayer
Allow multiple sources for testing data	Support online data by data downloader; support local data from device storage by local provider; support manipulated data by data manipulator
Support black-box testing	Replay data using the inter-process communication (Android Intent) in line with data collection of applications
Support white-box testing	Provide APIs for commands and audio data transmission using a code library in Java's JAR package, which the targeted application can import
Enable non-functional testing	Include evaluators for machine learning performance, processing speed and power consumption
Support testing on physical devices and emulators	Deploy the client as an Android application that runs with the targeted application; replay the data with a suitable number of threads (*i.e.*, the number of available CPU cores)



Target App Name: Fall Detection
Data Source Type: Accelerometer

Target App Name: Fall Detection
Data Source Type: Locations

Target App Name: App in Session
Data Source Type: Applications

Target App Name: Fall Detection
Data Source Type: Accelerometer
Status: pending

Target App Name: Fall Detection
Data Source Type: Locations
Status: pending

1. Select Target App Name
Fall Detection
2. Select Source Data Type
Accelerometer
3. Starting Timestamp
4. Ending Timestamp

☑ Tartget App Name: Fall Detection
Data Source Type: Accelerometer
Status: pending

□ Tartget App Name: Fall Detection
Data Source Type: Locations
Status: pending

Set Speed (e.g., 2.0)
10.0

Fig. 3. The TestAWARE client UI can be used to manage datasets and replay tasks.

To replay online data, the data downloader first downloads the data to its local storage. In contrast, KnowMe [9] simply fetches online data on the fly during the replay. Although downloading all the data may take a long time, developers can reuse the downloaded data if they need to replay data again. Another advantage of downloading is that the replay of downloaded data can stay uninterrupted by network faults.

4.1.2 Local Provider. To test applications with data already recorded on the device, developers can use a local provider to include such data in the replay. A typical scenario can be that developers first record some data on the phone using the targeted application or a middleware such as AWARE, and then they replay the data in the testing with the help of the local provider. Similar to data downloading, developers need to input file path, table name and timestamp period.

In Android application development, there are two options to copy local data: using a content provider or file stream. However, Android requires reading and writing permissions for each specific content provider when the data-processing application (e.g., TestAWARE) is developed. We are not able to know the details of these content providers since they are different across each testing task. Comparatively, it is easier for applications to read and write local data using a file stream, because Android only asks for a general permission "<uses-permission android:name= "android.permission.WRITE_EXTERNAL_STORAGE" />". TestAWARE copies local data files to its own folder. Then developers can include the local data during testing.

ALGORITHM 1: Concurrent Data Replay

```
INPUT: a dataset with nonempty data sources in the replay D={d[0], d[1], ..., d[n]}, speed multiple v
BEGIN:
1:  foreach data source d[i] do in parallel
2:      finished[i] ← false
3:      if d[i] is audio then
4:          goto line 20
5:      else
6:          set data instance I[current] as the first data instance of d[i]
7:          send I[current]
8:          if d[i] has next instance I[next] then
9:              set t as the time difference between I[current] and I[next]
10:             wait t/v
11:             I[current] ← I[next]
12:             goto line 7
13:         else
14:             finished[i] ← true
15:             while for all integer j, -1<j<n+1, finished[j] is true do
16:                 goto END
17:             end while
18:         end if
19:     end if
20:     set frame F[current] as the first frame of d[i]
21:     send F[current] via each channel of d[i]
22:     if d[i] has next frame F[next] then
23:         set s as the sample rate of d[i]
24:         wait 1/(s×v) second
25:         F[current] ← F[next]
26:         goto line 21
27:     else
28:         goto line 14
29:     end if
30: end for
END
```

4.1.3 Data Replayer. Unlike prior work [9], the data replayer of TestAWARE only replays localised data (i.e., downloaded data or data recorded on the device) to avoid the effects of unstable network connection. Also, localised data can be reused in different testing tasks. To carefully examine software details or accelerate the testing for longitudinal datasets, developers can specify replay speeds that are slower or faster than the real-time clock. The data replayer concurrently replays data from each data source using a suitable number of threads (i.e., equal to the number of CPU cores in the current environment). To synchronise sensor values, events, and audio files during replay, we have developed a concurrent algorithm whose pseudocode is shown in Algorithm 1.

We have implemented the data replay module using the *java.util.concurrent* package provided by Java 8. TestAWARE schedules all the data sources selected by developers using a *ScheduledExecutorService*, which considers each data source as a scheduling task. To replay data at the speed set by developers, this service schedules tasks periodically by recalculating the timestamp using the predefined replay speed. To maximise the efficiency in replay, the service detects the number of cores on CPUs and spawns the same number of threads. Once a task replays all the data from a data source, the service will finish the task and give resources to other ongoing tasks. For audio data, TestAWARE accepts Waveform Audio File Format (WAV format) with one channel. Unlike other sensory data and events, audio data is a stream of frames. For example, the sample rate of WAV is normally 44100 Hz. In practical replay, we found that sending every sample per time is a significant burden (44100 executions per second) for devices or emulators. To solve this problem, we create a buffer to send every 441 samples per cycle, which means 100 executions per second. This strategy substantially reduces the computational cost for audio data replay.

4.1.4 Result Recorder. Allowing for modifications of the source code in a white-box testing regime, the targeted application can output results to the TestAWARE client. The result recorder collects and stores this output on the device for further analysis. Developers can insert code into the source code to verify the internal behaviours of the targeted applications. For example, a line of text output code can effectively test whether the program runs into a branch under specific conditions. Developers can also record debugging output information or assertions to compare the actual values and expected values in the white-box testing. This feature is useful when developers want to evaluate machine learning modules. The result recorder can capture each prediction from the machine learning algorithm, and each ground-truth value from one of the data sources. Then, the machine learning evaluator compares the two groups of outputs and generates the analysis results. Furthermore, the result recorder can record real-time output for the processing speed calculator to analyse the real-time performance of the targeted application.

4.1.5 Machine Learning Evaluator. The accuracy of context recognition is an important requirement in the development of context-aware applications. However, evaluating machine learning algorithms in these applications is non-trivial. First, there are many kinds of the learning problems involved in these applications, including binary classification, multiclass classification, regression, hard clustering, soft clustering and so on. Second, datasets of these applications may contain data labels with different frequency. This is a challenge in the analysis of machine learning performance.

Suppose a dataset contains N raw data instances X = {x1, ..., xN}. The machine learning algorithm may directly take these data instances as input, serving as a function $f: X \rightarrow Y$, where Y is the output space of classification or regression. If the dataset also contains the ground truth y_i in Y, corresponding to each raw data instance x_i, the machine learning evaluator can easily assess performance by comparing the output of the algorithm $\hat{Y}$ and the ground truth Y given by the dataset.

However, it is conceivable that a machine learning algorithm of a targeted application may filter raw data or transform it into another form (*e.g.*, using feature extraction methods) to construct input which is not actually in the historical dataset. Additionally, the dataset may only contain Y_{sub} which misses some values from the complete ground truth sequence Y (*i.e.*, $Y_{sub} \subsetneq Y$). In these cases, it is challenging to correctly match each ground truth value of the dataset with the output of the machine learning algorithm. Hence, assuming the ground truth appears later than the triggering context (*e.g.*, when using Experience Sampling Method [15], to collect ground truth labels), we propose an output matching algorithm which operates together with the machine learning algorithm within the targeted application. Algorithm 2 shows the pseudocode of this algorithm. Note that developers have to indicate the maximal possible *time difference* between the last raw data instance and the ground truth label by specifying the delay tolerance T in seconds. Both $\hat{Y}$ and Y are saved by the result recorder.

ALGORITHM 2: Output Matching Algorithm

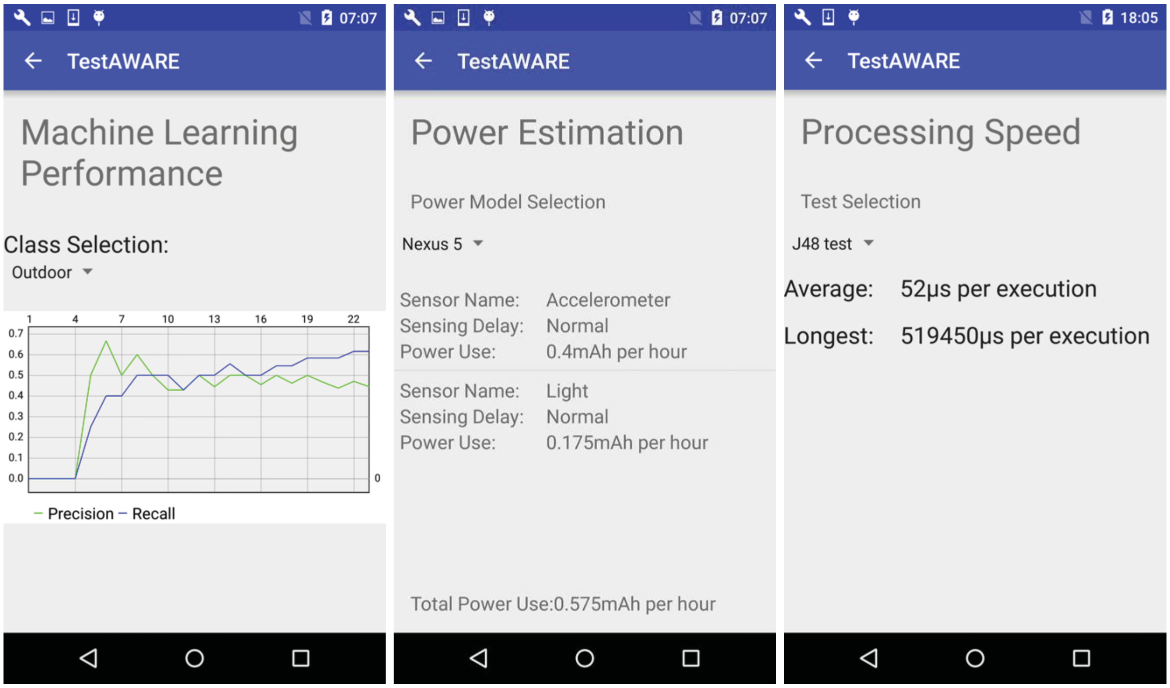
INPUT: *machine learning algorithm f, a dataset with raw data instances X = {x[1], ..., x[N]} and ground truth sequence Y (it is uncertain whether this sequence is complete), empty sequence $\hat{Y}$, delay tolerance T in real-time clock*
OUTPUT: *output value $\hat{Y}$, where $\hat{y_i}$ is considered to be corresponding to y_i in Y*

```
BEGIN:
while Y has next instance y[next] do
    set $T_y$ as the timestamp of y[next]
    while X has next instance x[next] do
        set $T_x$ as the timestamp of x[next]
        input x[next] into f
        if $T_x > T_y$ then
            add NULL into $\hat{Y}$
            break
        end if
        if f generates output $\hat{y}$ then
            if $T_y - T_x < T$ then
                add $\hat{y}$ into $\hat{Y}$
                break
            end if
        end if
    end while
end while
output $\hat{Y}$
END
```

For white-box testing, the machine learning evaluator uses the results captured by the result recorder to assesses accuracy. For classification tasks based on supervised or unsupervised learning algorithms, the machine learning evaluator measures prediction accuracy by comparing the output values to the ground truth in the dataset. For classification problems, the machine learning evaluator further measures the precision and recall of each class, as shown in Fig. 4a. For regression problems, the machine learning evaluator measures the mean absolute error (MAE) and mean squared error (MSE). The machine learning evaluator tracks performance during the whole period of data replay. Thus, developers can gain insights about the impact of increasing data amount on the machine learning performance.

4.1.6 Energy Evaluator. The energy evaluator, together with the machine learning evaluator, supports developers in balancing the tradeoff between power cost and the accuracy of context recognition. Similar to the sensor-trace method proposed in [19], developers can provide a sensor power model for a specific device. Based on this model, the energy evaluator estimates the power consumption by calculating the usage of each sensor with the sensing frequency in the replayed dataset. Fig. 4b shows an example of the output. If developers do not have such a model, they can refer to hardware manufacturers using AWARE.

4.1.7 Processing Speed Evaluator. For real-time sensing applications, computation time is a critical measure of an execution. For example, slow processing speed decreases the user experience of a context-aware UI. Hence, the TestAWARE library enables developers to measure and record the execution time of intensive tasks (e.g., machine learning) during white-box testing. Before and after the software module code in the targeted application, the developer can call related functions provided by the TestAWARE library. To measure multiple modules without conflicts, the developer can assign each measurement a label. After the test is complete, the TestAWARE client visualises the average and worst durations for the developer to view (Fig. 4c).



TestAWARE

Machine Learning Performance

Class Selection:

Outdoor

TestAWARE

Power Estimation

Power Model Selection

Nexus 5

Sensor Name: Accelerometer
Sensing Delay: Normal
Power Use: 0.4mAh per hour

Sensor Name: Light
Sensing Delay: Normal
Power Use: 0.175mAh per hour

Total Power Use:0.575mAh per hour

TestAWARE

Processing Speed

Test Selection

J48 test

Average: 52μs per execution

Longest: 519450μs per execution

(a) (b) (c)

Fig. 4. Screenshots of the TestAWARE client showing: a) machine learning evaluation results. The developer can choose which class label is of interest (*e.g.*, "Outdoor"), and view the precision and recall performance (y-axis). The x-axis shows the sequence of ground truth labels during the test; b) the energy consumption profiling; c) the processing speed profiling.

4.2 TestAWARE Code Library

The TestAWARE code library is a Java JAR package. Developers can import this package into the targeted application for white-box testing and non-functional testing. They can also use this code library as standalone to generate and record simulated context. The code library provides significant flexibility during testing because developers can insert the code anywhere inside the targeted application, or even in the code of other applications. The only requirement is that the code from the code library can be executed in the same environment (i.e., device or emulator) as the TestAWARE client. However, a drawback is that developers without Java coding skills (e.g., front-end designers) cannot make use of functions in the code library.

4.2.1 Command API. The command API offers controls provided in the UI of the TestAWARE client, including downloading data, replay data and stopping replay. By writing simple code snippets using the command API, developers are able to automate the processes of dataset download and launch/stop data replaying in white-box testing. Compared to using the client in black-box testing, the command API is more suitable for large-scale testing involving a large number of devices or emulators. To make use of the command API, developers must import the code library into the targeted application.

During the initialisation of the targeted application, the application's source code and the commands written by developers are executed together by the device or emulator. Then the commands send messages to the TestAWARE client for the data replay. Thus, the targeted application can start to process the contextual data sent from the TestAWARE client.

To check the internal behaviours of the targeted application, developers can write output commands and assertions inside the source code, for example, a loop or a branch. Output and assertion results are recorded by the result recorder, as illustrated before.

In non-functional testing, the command API provides commands to record machine learning predictions and the execution time in nanoseconds. Also, to estimate power consumption, developers can use the related commands to build a power model for a device.

4.2.2 Data Manipulator. The data manipulator enables developers to quickly simulate intended context (e.g., a fabricated application crash or battery low event) by generating synthetic (i.e., simulated) data. Except audio data, developers can manipulate any type of sensory data and Android event, using the functions provided in the data manipulator. Then, the data manipulator creates and stores these values in the local storage of TestAWARE on the device or emulator. In either black-box or white-box testing, the TestAWARE client can replay this synthetic data in the same way it replays historical datasets in local providers.

Using the data manipulator, developers can efficiently simulate their intended context without recording values from a realistic context or obtaining a historical dataset. For example, healthcare application developers can create a set of fake GPS coordinates to test whether their applications can find the nearest hospitals.

Unlike previous work [12] generating values using a specific tool, the data manipulator is a chunk of reusable Java code. This means that developers can conduct the simulation of intended context at any time in the testing. Developers may use the data manipulator to generate some new contextual data depending on the initial output of the targeted application. For example, when a smart home application detects that the user is coming home, developers can simulate different indoor temperatures to test whether the air-conditioner is turned on.

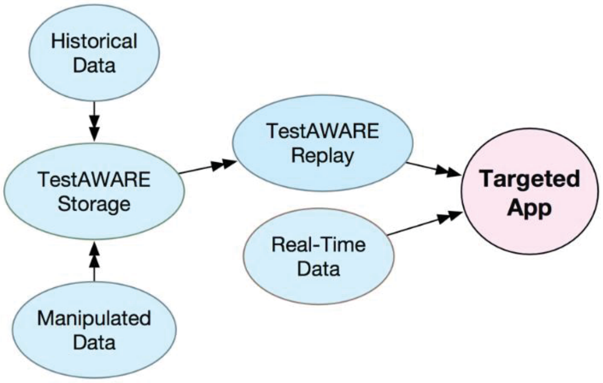


Fig. 5. Fusion of real-time, historical and manipulated data using TestAWARE in the testing.

4.2.3 Data Receiving API. The data receiving API is only for audio data replay. For AWARE-compatible hardware sensory data and raw Android events, the data collection and transmission process of the target application bears no difference to real-world scenarios. However, due to the specificity of microphone data stream, TestAWARE cannot override the microphone hardware to replay audio data. Consequently, the targeted application can only receive replayed audio streams using the data receiving API of the code library in white-box testing. Based on the data receiving API, developers can include audio as a data source in the replay to construct the intended context during testing.

4.3 Data Fusion Support

TestAWARE is able to replay datasets in the formats of AWARE-compatible hardware sensory data, raw Android events and audio files. Although developers can replay historical data to test applications processing different types of data, it may be still difficult to construct the intended context based on historical data only [17]. Fusing real-time, historical and simulated data is an effective way to construct context in more cases [17].

In TestAWARE, we implemented this idea for the construction of more sophisticated context, as shown in Fig. 5. Because TestAWARE does not change the way that the targeted application collects data, the targeted application still considers replayed data as real-time data. Thus, the targeted application can simultaneously receive data from TestAWARE and hardware sensors. If developers want to test applications using manipulated data, they can create and store such data before testing. Since manipulated data is replayed by the data replayer during testing, TestAWARE supports the fusion of the manipulated, historical and real-time data.

Hence, developers can leverage data fusion to construct sophisticated context by combining replayed data and real-time data. For example, to test a battery monitoring application that recommends the closest phone-charging location depending on the current location, developers can replay a fabricated battery low event and use the actual (i.e., real-time) location coordinates such as GPS. Note that developers should set the data replay speed to real clock (i.e., 1x speed) if they include real-time data into the data fusion.

5 EVALUATION

For the purposes of evaluation, we have decided to use both quantitative and qualitative measures. We first discuss the deployment of the output matching algorithm, and then we quantify the maximal speed of data replay. Finally, we evaluate the usefulness of TestAWARE in a user study with 13 professional mobile application developers.

5.1 Selecting Delay Tolerance

The key aspect of our tool is its data synchronisation during replay, and it is especially important to assess how this works with "messy" real-world data and labels. Our output matching algorithm requires a parameter delay tolerance T to match each machine learning output and the corresponding ground truth label. Developers have to specify this tolerance empirically in their code snippet, depending on the characteristics of the dataset and the process of context recognition.

For example, in the field study in [29], the machine learning problem is to predict, every time the user unlocks their phone, whether the user is going to start a new task or continue their previous task. In this dataset, the ground truth labels ("NEW_TASK" vs. "CONTINUE_TASK") were collected by explicitly asking the users after they had unlocked their phone. As a result, the ground truth labels appear after the actual unlocking event, and this delay varies considerably due to inconsistent human labelling (Fig. 6). Hence, the ground truth labels have imprecise timing in this case.

In addition, the performance of the classifier (i.e., a constant classifier) during our testing can vary. For instance, we measured the time needed for our classifier to determine whether a user who just unlocked the phone intends to start a new task, or continue a previous task. The time needed by our classifier is shown in Fig. 7. This graph was constructed by visualising the values recorded by TestAWARE.

By comparing Fig. 6 and 7 we find that there is a discrepancy between when the ground truth label appears in the historical data, and when our classifier is able to validate its machine learning assertion during testing. We do not want to penalise the application for getting the timing of its prediction wrong, and therefore our Algorithm 2 allows for a Delay Tolerance to account for this discrepancy. In this case, we find that our classifier can take up to 0.678 ms to execute. At the same time, we find that some ground truth labels appear with delay of up to 100 seconds. Therefore, to correctly match our application's machine learning output and the ground truth labels, we would set the delay tolerance to be slightly longer than the majority (e.g. 95%) of the observed time difference between our application's machine learning output and the ground truth labels. This is approximately 5 seconds in our case.

5.2 Maximal Data Replay Speed

In the testing of context-aware applications with longitudinal data collection (e.g., chronic disease tracking), it is necessary for developers to replay a large amount of historical data in a short period. Hence, we quantify how fast TestAWARE can replay data. We selected 6 smartphones, 6 tablets and 4 PC-based emulators to replay sensor, event and audio data at the fastest speed. In the sensor data replay, we used one thread to replay 10000 accelerometer readings, which is equivalent to about 1 minute of sensing with the highest fidelity. In the event data replay, we used one thread to replay 10000 ACTION_BATTERY_LOW events, which typically occur when the phone battery becomes low. For audio replay, we used one thread to replay a file in WAV format with one channel of 16-bit stream (sampling rate = 44.1 kHz, samples = 169529220).

5.2.1 Data Replay on Smartphone. First, we investigate the maximal replay speed on 6 off-the-shelf smartphones: LG Nexus 5 with Android 5.1.1, Samsung S6 Edge with Android 6.0.1, Motorola Moto G2 with Android 5.0.2, 2 × Motorola Moto G1 with Android 4.4.4 (G1-1 and G1-2), Motorola Moto G1 with Android 5.1 (G1-3).

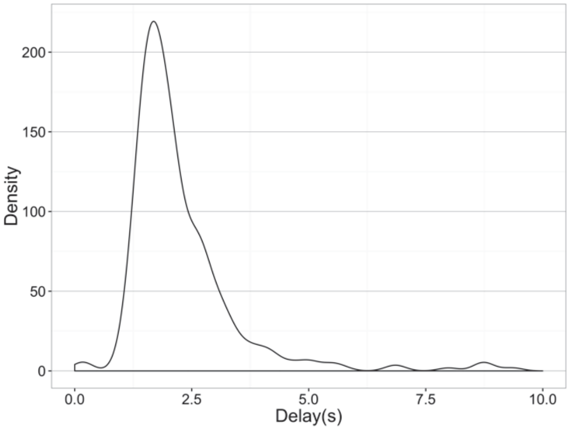


Fig. 6. Delay between unlock event, and ground truth label in our historical data.

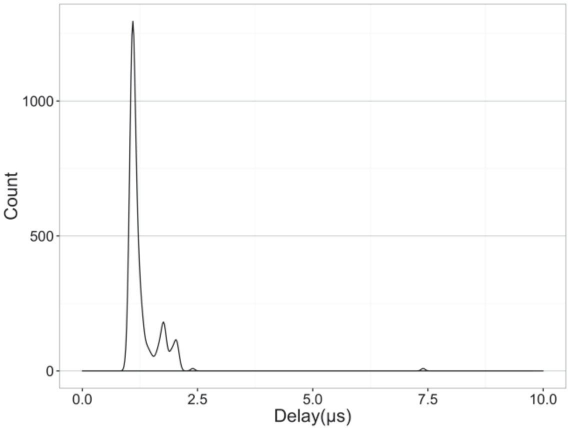


Fig. 7. The variation in the time needed by our classifier to make an inference.

Fig. 8 shows the maximal replay speed of audio on smartphones, in multiples compared to the original playback speed of the audio file. Results show that all these devices performed very differently. They were able to accelerate the replay speed to the range of 4.46 - 13.77 multiples. The best performance (13.77 on Nexus 5) was approximately triple the worst (4.46 on S6).

Fig. 9 depicts the maximal replay speed of sensor and event data on smartphones. Similar to audio replay, results show that devices performed very differently. All the handsets can replay at least 1000 instances of either sensor or event data per second. Also, we observe that all the handsets replay events faster than sensor data. Except S6, the replay speed of events is significantly higher than that of sensor data.

5.2.2 Data Replay on Tablets. Next, we measured the maximal replay speed on 6 off-the-shelf tablets: Samsung Galaxy Tab Pro 8.4 with Android 4.4.2, 3 × Samsung Galaxy Tab4 10.1 with Android 5.0.2 (10.1-1, 10.1-2 and 10.1-3), 2 × Lenovo TAB3 7 with Android 6.0 (TAB 3 7-1 and TAB 3 7-2).

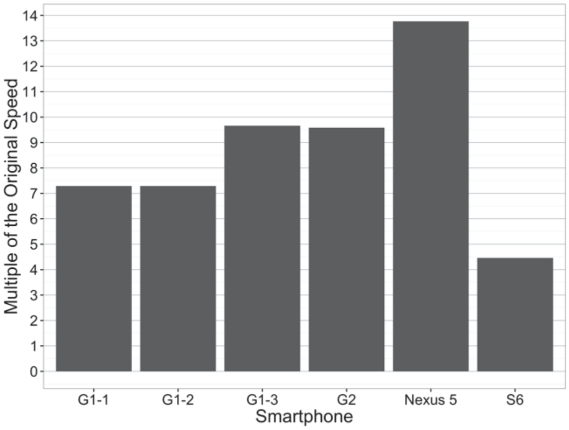


Fig. 8. Maximal speed of replaying audio files on smartphones.

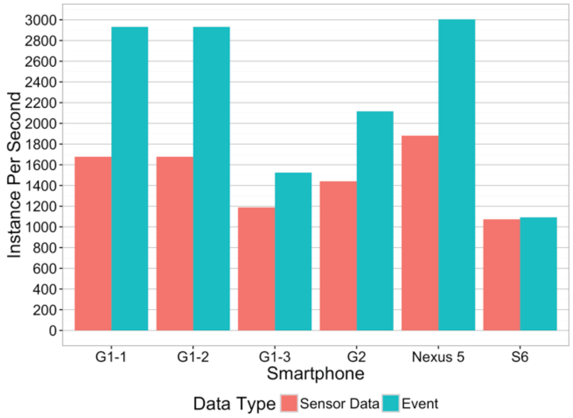


Fig. 9. Maximal speed of replaying sensor data and events on smartphones.

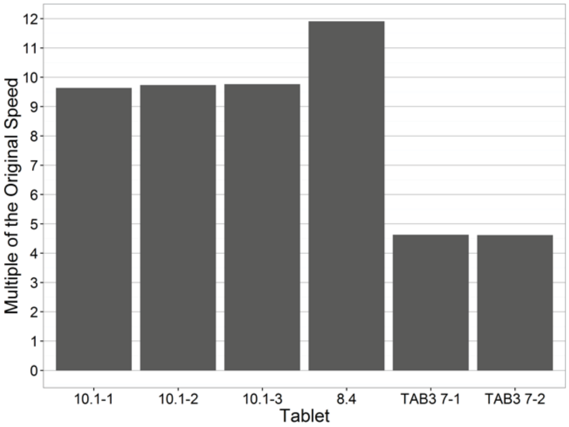


Fig. 10. Maximal speed of replaying audio files on tablets.

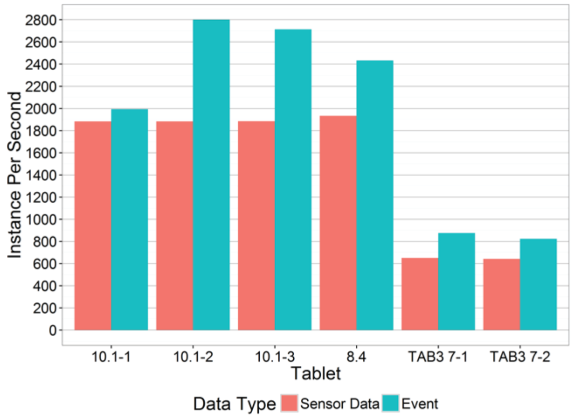


Fig. 11. Maximal speed of replaying sensor data and events on tablets.

Fig. 10 shows the maximal replay speed of audio on tablets, in multiples compared to the original playback speed of the audio file. Results show that the performance significantly differed. These tablets were able to replay the audio file within the speed range of 4.61 - 11.91 multiples. Tab Pro 8.4 performed the best with 11.91 multiples. Identical models of Tab4 10.1 and TAB3 7 had similar performance.

Fig. 11 presents the maximal replay speed of sensor and event data on tablets. Results show that the performance of replaying sensor data and events greatly varied among different models of tablets. TAB3 7 had the lowest performance with less than 700 instances of sensor data per second and 900 event instances per second. On other models, the performance was more than doubled: at least 1800 instances of sensor data per second and at least 1900 event instances per second.

5.2.3 Data Replay on Emulators. Finally, we measured the replay speed on a smartphone emulator (Nexus 5 with API 22) and a tablet emulator (Nexus 10 with API 22) using 2 PCs: Mac Mini with Intel i7 2.3GHz and MacBook Pro with Intel i5 2.7GHz. We allocated 2GB RAM, 500MB Android VM heap size and only one CPU to each emulator.

Fig. 12 shows the maximal replay speed of audio on emulators, in multiples compared to the original playback speed of the audio file. Results show that all these emulators performed similarly. They achieved the maximal replay speed to the range of 19.31 - 21.88 multiples. The best performance (21.88 on Nexus 10 MacBook) was slightly better than the worst (19.31 on Nexus 5 Mini).

Fig. 13 presents the maximal replay speed of sensor and event data on emulators. Results show that all the emulators can replay at least 3000 instances of sensor data per second. For the replay of events, all the emulators can achieve a much higher speed: over 6000 instances per second. Similar to audio replay, the performance of replaying sensor data and events slightly varied across emulator types (smartphone and tablet) and PC models.

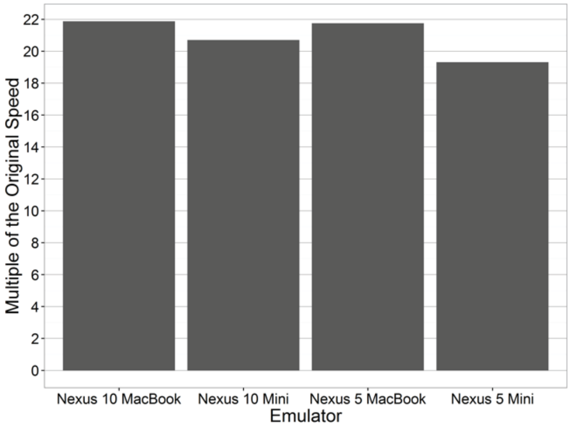


Fig. 12. Maximal speed of replaying audio files on emulators.

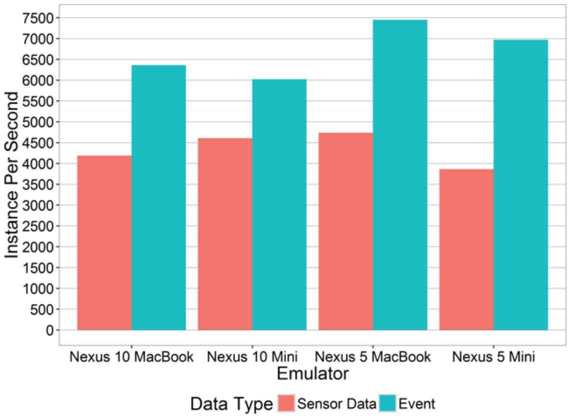


Fig. 13. Maximal speed of replaying sensor data and events on emulators.

5.3 User Study

To investigate the usefulness of TestAWARE, we conducted a user study with two real-world testing tasks. To select a suitable sample size of participants, widely accepted practices suggest 12 to 16 users [5]. Hence, we recruited 13 participants (3 females, 10 males, average age 25.8, average professional years 2.2). They were recruited through social media. All participants are professional programmers working in the industry of mobile software development for at least 1 year. The participants at least hold a bachelor's degree in computer science or related areas. Each participant was compensated with a gift card equivalent to 10 EUR.

All participants were interviewed individually in our lab. For each interview, we first introduced TestAWARE to the participant. We described and explained the features of the mobile client on a smartphone and an emulator running on a PC. Then we showed them a demonstration where an Android application receives replayed data from the TestAWARE client both on the phone and emulator.

Next, we completed a hands-on training session. Participants were instructed on how to use all the features of the TestAWARE client and code library. They were provided with available datasets, a template Android application project, API syntax and code examples. This training session was completed within one hour.

After the training was complete, participants were given a real Android application project for two testing tasks. The targeted application collects sensor data to predict applications to be used, and is implemented based on the work described in [26]. The predictions are shown on the screen as application icons. Before the study, we tested the implemented application carefully. Then we intentionally inserted a bug to the source code as a functionality flaw. Regardless of predictions, this bug always wrongly displayed a group of icons. The first testing task was black-box testing. The functionality of this application was introduced to participants. Participants were not familiar with the application's inner implementation structure and details. Participants were asked to test the application using the TestAWARE client only. The second testing task was white-box testing. Full source code and reference publication [26] were given to participants. In this task, participants were allowed to use the TestAWARE code library with the client. Participants were asked to not only test the application functionality, but also assess its non-functional properties including machine learning performance, power consumption and processing speed. Participants were given unlimited time to finish the tasks.

After both testing tasks were completed, each participant answered a questionnaire with nine questions relating to the testing tasks and our tool. For each question, participants were instructed to give a score on a scale from 0 (strong disagreement) to 100 (strong agreement). During the subsequent interview, we also asked each participant to explain their scores.

5.3.1 Results. The results in Table 3 show the numbers of participants according to their states towards the flaw for each stage. After black-box testing, 7 developers reported the flaw about icons because they found that the icons did not change during the whole process of data replay. However, 6 developers could not report this flaw because they thought that, even if the icons did not change, they may reflect correct predictions corresponding to the data replay. After white-box testing, all developers reported the flaw because they used run-time assertions to check the details of the program. Based on the results of assertions, they all correctly located the bug in the source code.

Table 3. Results of two testing tasks.

Number of Participants	Before Testing	After Black-Box Testing	After White-Box Testing
Reporting the flaw	0	7	13
Not reporting the flaw	13	6	0

We compared the states before testing and after black-box testing using Pearson's chi-squared test. We found that the black-box testing with the mobile client offered a significant help for testers to spot a flaw (P=0.008). In addition, we compared the effectiveness of our tool in the black-box and white-box testing. The results indicate that participants received a significantly greater support when using the TestAWARE client and code library in the white-box testing (P=0.02).

Question 1: Does data replay from TestAWARE help the testing in the two tasks? (average score: 90.8, median score: 90, standard deviation: 6.7) This question investigated whether our tool, in a practical manner, helps developers conduct both black-box and white-box testing of context-aware applications with data replay. The scores indicate that the participants largely perceived the necessity of data replay in the testing of context-aware applications, and that they identified a lack of tools to conduct data replay for this kind of testing. All participants considered data replay as a workable way to simplify the testing by avoiding collecting new samples of context in every round of testing. They also had unique opinions. In terms of testing cost, P2 stated: *"Data replay reduces the cost of data collection in industrial testing, without loss of fidelity"*. Highlighting the support for automation, P5 stated: *"To automate the testing, testers must have a tool for data replay"*. Regarding the likelihood to identify a bug, P11 and P13 commented that data replay can reproduce bugs to help testers understand the causes: *"Without data replay, it is hard to reproduce a bug which was already detected in the past. If a detected bug cannot be reproduced, developers can hardly find the reason and fix the bug"* (P13). The low standard deviation shows that the participants had a little divergence when judging the magnitude of usefulness. P8 giving a score of 80 stated: *"Data replay is able to make the application run in testing. Testing it directly in real-world context can also be a way to go"*.

Question 2: Does the TestAWARE mobile client help the testing in the first task? (average score: 80.2, median score: 85, standard deviation: 15.1) This question was to check whether the TestAWARE mobile client supports developers to conduct the black-box testing. The scores show that the participants considered the mobile client as a useful component of our testing tool. The majority of participants commented that the client is easy to use and does not require testers to write scripts. However, the high standard deviation indicates that participants judge its usefulness differently. Regarding the search for bugs, P1 stated: *"The client only replays the data for the tested application. It is hard to detect a bug because testers do not know whether the data is successfully received. And testers do not know the correct output from the tested application"*. In terms of testing coverage, P13 stated: *"Testers do not know how completely the application is tested. A number of bugs may be missed"*. P2, P5 and P11 stated: *"It does not help you locate the bugs"*. P12 stated: *"It helps testers who do not write code. But many bugs can be missed"*.

Question 3: Does the TestAWARE code library help the testing in the second task? (average score: 91.2, median score: 90, standard deviation: 7.5) This question was to investigate whether the TestAWARE code library supports developers to conduct the white-box testing. The scores show that participants perceived the substantial usefulness of the code library, with little divergence indicated by the low standard deviation. Most participants commented that the code library enables the automation and more careful examination in the testing. P1 stated: *"The commands can automate a large number of testing rounds"*. P8 stated: *"The code library helps the testing on not only high levels, but also the level of unit testing and integration testing. It can examine every step of programs. It also locates the bug in the code"*. P6 stated: *"It is easy to debug when a flaw is detected using the code library"*. P12 stated: *"The code library allows testers to match the output of software and different input"*. However, they argued that it takes long time to conduct testing using the code library. P8 stated: *"The limitation is that testers use a lot of time to write testing scripts"*. Accordingly, P13 suggested a solution: *"In practice, we may test only important components using this way, rather than all details"*.

To confirm the effectiveness difference of our tool in the black-box and white-box testing, we compared the scores of Q2 and Q3. First, we tested the normality of each distribution using the popular Shapiro-Wilk test. We found that the scores of Q2 are not normally distributed (P=0.010), but those of Q3 are (P=0.070). Hence, we used Mann–Whitney U test to verify the difference. The test identified a significant difference between the two sets of scores (P=0.041).

Question 4: Does the assertion function help the testing in the second task? (average score: 89.5, median score: 90, standard deviation: 10.1) This question was to investigate whether the run-time assertion function of the TestAWARE code library provides help in the white-box testing. The high scores indicate that the participants appreciated the importance of assertions in the testing. All participants agreed that assertions accurately located the bug in the code. P2 stated: *"Using run-time assertions is efficient and accurate. They are better than breakpoints because they require applications to run only once"*. However, the slightly high standard deviation indicated some limitations of run-time assertions: *"Given a bug location, it still requires some logical analysis to fix the bug"* (P10); *"Although a bug is located, understanding the whole scenario causing the bug is sometimes a complex task"* (P13).

Question 5: Does the machine learning evaluator help the testing in the second task? (average score: 82.7, median score: 90, standard deviation: 19.4) This question aimed to investigate whether the machine learning evaluator of our tool helps developers in analysing the machine learning performance of the application. The scores reveal a general endorsement among participants. P1 stated: *"It quickly gives an initial performance summary to testers"*. P11 stated: *"It reflects the quality of code which uses machine learning algorithms"*. Comparatively, some participants perceived only limited usefulness, causing the high standard deviation: *"The tool evaluates machine learning using several simple measures. It may produce inaccurate evaluation results"* (P6); *"Common testers may not pay much attention to machine learning performance. Also, users may not care the accuracy of machine learning results"* (P12).

Question 6: Does the power estimator help the testing in the second task? (average score: 83.2, median score: 90, standard deviation: 23.2) This question was to investigate whether the power estimator helps developers to assess the power consumption. The scores show that the participants have the need to estimate the power consumption of context-aware applications. Regarding industrial mobile application development, P1 stated: *"The assessment of power use is indeed a process in mobile software production. An estimation is useful for testers to refer to"*. P10 stated: *"Estimation of power use from different sensors can help testers balance the data collection and battery preservation"*. However, several participants argued that the estimation may have large errors, resulting in the high standard deviation. P5 and P7 stated: *"The tool has only estimation. It cannot measure the actual power consumption of practical usage"*.

Question 7: Does the processing speed evaluator help the testing in the second task? (average score: 86.3, median score: 90, standard deviation: 10.4) This question aimed to investigate whether the processing speed evaluator helps developers in analysing the efficiency of certain procedures in the application. The scores reveal the considerable usefulness perceived by participants. P7 stated: *"Response time is an essential index of user experience for mobile applications, especially for Android"*. In terms of optimisation of applications, P6 stated: *"It helps testers find the bottlenecks which testers may try to optimise"*. P11 stated: *"Testers can compare different implementations and find the best one"*.

Question 8: Is the maximal replay speed of audio, sensor and event data sufficient to in the testing? (average score: 94.8, median score: 98, standard deviation: 5.7) This question was to investigate whether the maximal speed of data replay satisfies the need of developers in the testing. As indicated by the high scores and low standard deviation, all the participants agreed that the maximal replay speed suffices for efficient testing. P3 stated: *"It is fast enough. It reduces the testing time compared to testing in the real context"*. Regarding testing applications with longitudinal data collection, P11 and P12 stated: *"The high speed is enough to quickly complete a test if the data was collected across a long time"*.

Question 9: Is it a useful feature of TestAWARE to support both physical device and emulator? (average score: 96.9, median score: 100, standard deviation: 6.0) This question was to investigate whether TestAWARE helps developers in the testing by supporting both physical device and emulator. All participants agreed to the usefulness of the environment support, producing the high scores and low standard deviation. In terms of processing speed measurement, P1 and P4 stated: *"Using real devices to measure process speed is reliable"*. Regarding the convenience of development and testing, P6 stated: *"I like to use emulator for development and testing due to the convenience"*. Similarly, P8 stated: *"Emulators are easy for testers to automate testing"*. In terms of compatibility testing, P11 and P13 stated: *"It is necessary to test applications on real devices for checking compatibility on different hardware and OS"*.

6 DISCUSSION

Our overarching goal is to enable developers to test context-aware applications in laboratory settings. There are many reasons why this goal is desirable, including the ability to more systematically test the software, to enhance the reproducibility and replicability of test results, and primarily to reduce costs associated with participant-driven trials. Our tool enables developers to conduct automated tests in laboratory settings, thus greatly reducing the need (or offsetting the need) for user trials, at least in early stages of development of context-aware software. Clearly, we do not suggest that user trials are not needed, especially in terms of user interface design, but certainly automated testing can help in identifying functional and non-functional flaws.

6.1 User Study Findings

We begin by summarising the findings from our interviews with developers. The interviews first sought to investigate whether our tool can practically help developers test context-aware applications with data replay. Participants confirmed the necessity of data replay in the testing of context-aware applications, and they confirmed the lack of tools to conduct data replay for this kind of testing. Specifically, participants highlighted the simplicity of conducting multiple rounds of testing with TestAWARE. They also considered that data replay is effective in cost reduction, automation and bug reproduction. We also inquired the usefulness of the TestAWARE client in the black-box testing. The participants considered the client as a useful component because it is easy to use and does not require testers to write scripts. Additionally, participants considered the code library as a useful component of the testing tool because of automation support, bug localisation and careful examination from low levels to high levels.

Regarding the testing of non-functional requirements, the participants felt that our tool supported them effectively for the analysis of machine learning performance, power consumption and processing speed. However, some participants argued that these three aspects are of different usefulness to them. They considered that processing speed is more useful than machine learning performance and power consumption estimation, due to the importance of user experience. In terms of the maximal speed of data replay, all participants agreed that the speed is sufficient for testers to conduct testing efficiently. Finally, all participants noted that support for both physical devices and emulated devices is crucial for testing. All the participants preferred the freedom to select testing environments between the two options. Some stated that, for convenience, they would only use emulator if they have data for replay. Some thought that they would always include physical devices in testing, not only for measuring processing speed, but also for the assurance of software compatibility.

To quantitatively evaluate our tool, we first statistically compared the effectiveness of our tool in the black-box and white-box testing. From the results of two testing tasks, we observed that our tool successfully supported testers in both the black-box and white-box testing. Furthermore, testers receive significantly greater help when the code library was used with the client. From the interviews, we confirmed this observation by the significant difference in the score distributions from the two related questions. Within the white-box testing support, the scores show that the assertion is a critical and useful feature of our tool. From the scores of the remaining questions, we confirmed the necessity of data replay and testing non-functional properties including machine learning performance, power consumption estimation and processing speed. The scores show that processing speed is of the highest usefulness among non-functional properties. Regarding the maximal speed of data replay, the scores indicate that testers can conduct testing efficiently with adequate replay speed. Regarding testing environments, the scores show that testers need the flexibility across real devices and emulators.

6.2 Data Replay for Testing

By replaying heterogeneous data including AWARE-compatible hardware sensors, Android events and audio files, developers can use TestAWARE to test the context-aware modules of mobile applications. The results of our user study suggest that data replay using TestAWARE simplifies the testing of context-aware applications. Compared to prior work such as MobiPlay and KnowMe, TestAWARE supports audio from microphone as an additional data source. Furthermore, TestAWARE allows developers to construct the intended context by creating simulated/synthetic data, or using data fusion to combine real-time data and replayed data. This is a key characteristic that can reduce the efforts and cost in data collection or dataset download. For example, manipulated data can be used to construct uncommon contexts, such as app crashes and human falls.

When replaying data using TestAWARE, developers can set a replay speed that is faster or slower than the real-time clock. This can accelerate the testing with longitudinal datasets. Also, by replaying data at a slow speed, developers can carefully verify the execution details of context-aware modules, such as step counter. If the data fusion uses a real-time data source, the replay speed must be set to 1 (i.e. real-time).

TestAWARE enables developers to manage data replay using either its mobile client (i.e., black-box testing) or the code library (i.e., white-box testing). As indicated by our interviews, the mobile client plays a crucial role in the testing when testers do not have programming skills. Challenging as programming is, the code library enables automated testing and allows developers to check low-level details of software code by recording the output of applications. In addition, the code library provides the only access to the testing of non-functional properties such as machine learning performance, energy consumption and processing speed.

Finally, TestAWARE can operate on either physical devices or device emulators. This offers developers the freedom to select suitable environments for the testing. As suggested during our interviews, given access to data for replay, it is convenient to use solely an emulator in testing. However, for the assurance of software compatibility, it is necessary to run applications and TestAWARE on physical devices. Also, if developers conduct tests with data fusion involving both replayed data and real-time data, they have to use physical devices and set the data replay speed to 1 (i.e., as fast as the real-time clock). According to our results, the maximal data replay speed of audio, sensor and event data on emulators is significantly faster than that on physical devices. This means that emulators are more suitable than physical devices when replaying data of longitudinal datasets.

6.3 Testing Non-functional Properties

By modifying the application source code in white-box testing, developers can use TestAWARE to record application output for debugging as well as testing non-functional properties, which is an important task for mobile applications due to the limited computational resources and battery life on mobile devices [7]. TestAWARE provides an analysis of the machine learning performance, power consumption and processing speed of the targeted application. Developers can make of these features to optimise applications for non-functional requirements.

As indicated in our interviews and prior work [19], these properties may be of different importance to specific applications. For example, compared to processing speed, power consumption is not an important measure for an application processing audio since developers normally cannot change the configuration of the microphone. It is worth noting that TestAWARE does not force developers to write testing code for all features. According to the mechanisms of their context-aware applications, developers can include or exclude specific modules from the code library when writing the code for the white-box testing.

6.4 Implications for Testing Mobile Context-Aware Applications

This section summarises the impacts of our work to the testing of mobile context-aware applications, in terms of conducting effective testing using our tool (i.e., guidelines for testers) and designing similar testing tools (i.e., guidelines for testing tool developers/researchers).

6.4.1 Heterogeneous Data. Some mobile context-aware applications rely on only one kind of contextual data. Testers can easily find tools listed in Table 1 to replay or manipulate sensory data or events. For mobile context-aware applications collecting heterogeneous data, testers have much fewer selections. To test mobile applications based on audio data, testers can only choose TestAWARE. Unlike sensory data and events, common testers cannot write scripts to generate audio files for testing, since they are not human-readable. Yet, to test mobile context-aware applications, TestAWARE is the only tool supporting sensory data, events and audio, enabling a new type of testing which is not covered by prior work. Testers can set a speed for data replay, conducting tests which are faster (e.g., to save time when testers use a longitudinal dataset) or slower (e.g., to carefully monitor the applications) than the real-time clock.

However, testers should have a suitable dataset for TestAWARE to replay. If testers are able to obtain such a dataset (e.g., by collecting application usage data from a sufficient number of users), tools, such as TestAWARE, enable the examination of dynamic behaviours and reduce the efforts in the testing of mobile context-aware applications, as discussed in similar work [22]. With such a dataset, testers can easily conduct regression testing in the future if they change the targeted application. Similarly, testers may reuse datasets to test other applications which process the same types of data as the targeted application.

When synchronising these three kinds of data in data replay, designers of testing tools should use:

1. the timestamps of sensory data and events. Typically, context management middleware (e.g., AWARE [10]) imprints a timestamp on each entry of sensory data and events. Designers should notice the format of timestamp, for example, in the unit of milliseconds or nanoseconds.
2. The frame numbers of an audio file. Audio files do not contain timestamps, but their sample rates are stable over time. For example, an audio file with 44.1kHz sample rate has 44100 samples per second. For audio files in a single channel (smartphone/tablet microphones normally generate audio in one channel), one frame contains one sample. Hence, in line with Algorithm 1, designers can use frame numbers to synchronise audio and other two types of data. To improve replay efficiency, designers can apply a buffer to replay a number of samples per time.

6.4.2 Multiple Sources of Testing Data. To encourage the reuse of datasets for faster and cheaper testing, it is important for testing tools to support multiple sources of testing data. However, none of existing tools allow testers to import a dataset from different sources. This issue lowers the chance of dataset reuse, possibly causing more labour, time and cost in testing. For example, rather than downloading an existing dataset from the Internet, testers must record a new dataset for MobiPlay [22] to replay in testing. Comparatively, using TestAWARE, testers can import data from online databases, local device storage (e.g., data files in AWARE format) and the manipulator (e.g., creating a dataset with manipulated values). Furthermore, testers can also lean on the record-and-replay testing method because TestAWARE uses the data format of AWARE. That is, testers can first record data using AWARE and then replay the recorded data using TestAWARE.

To support multiple data sources, developers of testing tools may follow the same implementation of TestAWARE. Data from online databases can be downloaded by common query statements such as SQL. Unlike directly downloading a file, this method has sufficient safety under data encryption and access control. Access to local device storage (e.g., record-and-replay) is a trivial task for mobile developers. To implement a data manipulator, developers can refer to the data format of the testing tool. To improve dataset reuse, they should use the data format of a popular context management middleware that is already used by previous projects and studies, rather than inventing a new format. Moreover, if they refer to an open-source context management middleware, they can save significant time by reusing its code of data management.

6.4.3 Black-box Testing. Existing testing tools mostly emphasise the support of black-box testing. However, when using some of them, testers must modify the way the targeted application receives data, such as reading data from files on MobiPlay [22]. TestAWARE also implements this feature using a mobile client running together with the targeted application. Using TestAWARE, testers do not have to change the targeted application (except applications based on audio data, because they listen to microphone. For audio replay without loss of audio quality, these applications should receive audio data from another channel.), because TestAWARE sends the data using Android Intent with the same data format as AWARE. To conduct black-box testing on applications, testers can efficiently switch off AWARE data collection (i.e., the normal way applications collect data) and start data replay of TestAWARE. Applications, ignorantly, receive the replayed data from the same channel. Hence, TestAWARE does not require coding skills from testers for black-box testing. A drawback of TestAWARE, as well as other tools, in black-box testing is that it only replays data for applications to process and does not explicitly report or locate bugs. Testers themselves should draw conclusions. If testers are sceptical about the application, they should further conduct white-box testing.

To enable black-box testing, developers of testing tools may adapt the mobile client part of TestAWARE. Importantly, to encourage testing conducted by testers without programming skills, developers should not modify the original way the targeted application collects data, unless there are technical difficulties (e.g., applications based on audio data must use another channel to receive replayed data, rather than listening to microphone). Regarding the help in bug detection, black-box testing can hardly explicitly report or locate bugs because of the ignorance of internal details. Developers may implement CPU and memory monitoring features which can provide indirect evidence for testers to infer the existence of some flaws such as memory leakage.

6.4.4 White-box Testing. For general mobile context-aware applications, none of extant tools support white-box testing. Using TestAWARE, testers can conduct white-box testing by its APIs of the code library provided in a Java's JAR package. To check the internal details of applications, testers can output values or apply assertions (i.e., comparing the actual value and the expected value) on values inside the source code. Using the code library of TestAWARE, testers can record the dynamic behaviours of applications at run time which lead to accurate bug localisation and careful examination from low levels to high levels, as demonstrated in our user study. Furthermore, the code library allows testers to automate dataset management (i.e., downloading data, replaying data and stopping replay), indicating that testers can perform large-scale testing on a large number of devices or emulators. To test applications based on audio data, testers must change the way applications receive data due to the speciality of microphone data stream. Based on the audio data receiving API in the code library, testers are able to include audio as a data source in the data replay.

As highlighted by our work, white-box testing is a promising and useful feature which developers of testing tools may implement in future work. They can employ the mechanism of the code library, allowing testers to insert testing code inside their applications. To support the examination of applications' internal behaviours, developers should design the commands and a result recorder for value output and assertions. For testing automation, developers should create commands for the controls of data management and replay. For the cases where testers have to change the way applications receive data, it is necessary for developers to provide corresponding data receiving commands for these applications to collect data from a new channel. Beyond TestAWARE, developers may include code coverage criteria, such as statement coverage and decision coverage, in their tools to assess the completeness of white-box testing.

6.4.5 Non-functional Testing. A small number of extant tools are specialised for testing non-functional properties of mobile applications, including machine learning performance, processing speed and power consumption estimation. However, they do not provide any features for functional testing at the same time, causing testers to spend more time and efforts on a complete testing process using different tools. This issue raises considerable challenges for regression testing which happens after every new change in software. To avoid this problem, testers can use TestAWARE which is able to conduct non-functional testing in parallel with functional testing. For the evaluation of machine learning performance and processing speed, testers can insert corresponding commands which record machine learning results and execution timestamp at the run time of programs. For the estimation of power consumption, testers need to provide a sensor power model for a specific device because each sensor on different devices varies in power use. TestAWARE records all the results on the device storage and reports them via the UI of the mobile client. Testers can also analyse each entry of the results by reading the data from the device storage.

Developers of testing tools should provide similar support for non-functional testing. For machine learning performance, they may implement more measures (e.g., Receiver Operator Characteristic (ROC) Area) beyond precision and recall. To achieve this, developers need to collect more run-time information from different machine learning algorithms. When recording timestamps for the measurement of processing speed, TestAWARE only allows testers to provide a name of the test. To support better evaluation of processing speed, developers can apply an assertion feature, which can be used by testers to compare the actual and the expected execution time. With this feature, testers can better find the bottlenecks in the programs in terms of processing speed. For the measurement of power consumption, developers can adapt the estimation method used by TestAWARE. Furthermore, they can attempt to monitor the actual power use of the targeted application on device battery. However, this is difficult because the operating system and testing tool also consume power at the run time of the targeted application. Developers could investigate how to accurately distinguish the power consumption from different sources.

6.4.6 Testing on Physical Devices and Emulators. To examine properties such as compatibility and processing speed, it is necessary for testers to run their applications on physical devices. Using TestAWARE, testers can run tests for their applications running on any smartphone and tablet with Android OS, which is the most popular mobile platform in the market. Also, supporting physical devices enables the data fusion of replayed data and the data collected by sensors in real time. For example, to test location-based applications, testers can use real-time GPS coordinates (i.e., using the GPS location from AWARE middleware) together with replayed data for other sensors (i.e., using TestAWARE data replayer). In some cases (e.g., physical devices do not have an OS version required in testing), testers need emulators to conduct testing. With TestAWARE, testers can also perform data replay and functional/non-functional testing on emulators. Regarding non-functional testing on emulators, the measurement of processing speed depends on the PC hardware of emulators, providing limited usefulness. Based on the results of our experiments on the maximal replay speed of data replay, testers should be aware that PC-based emulators can replay data significantly faster than actual smartphones and tablets. Testers can take advantage of PC-based emulators in testing scenarios where testers aim to efficiently replay longitudinal datasets in short testing time.

To enable the support of physical devices and emulators, developers of testing tools can simply implement their tools as applications on the targeted OS platform. For Android, developers can reuse the implementation of TestAWARE. As Android is implemented in Java, developers can make a Java's JAR package as a code library for the API commands of white-box testing. For other platforms, developers should use the corresponding programming languages for the implementation of testing tools and code libraries (e.g., Swift for iOS, C# for Windows Phone). Note that the mechanism of data replay functions may be restricted by different mobile platforms (e.g., the sandbox policy of iOS higher than 7.0 bans inter-process communication).

6.4.7 Benefits of using TestAWARE. In summary of the above aspects, we identify a number of unique features of TestAWARE that can benefit testers in testing mobile context-aware applications, beyond extant testing tools. In Table 4 we demonstrate the benefits brought by each unique feature of TestAWARE.

With TestAWARE, testers can conduct new types of testing: testing applications collecting audio data, and white-box testing. When initialising synchronised replay with any of sensory, event and audio data, testers can define the speed of data replay for tests which are faster or slower than the real-time clock. To alleviate the difficulties of finding testing data, TestAWARE improves the chance of dataset reuse. To reduce the time and cost in testing, TestAWARE automates and simplifies the data preparation phase, the process of black-box testing and non-functional testing. Also, to save time in white-box testing and non-functional testing, TestAWARE can conduct these two at the same time by providing a code library with API commands.

Table 4. Benefits of using TestAWARE.

Feature	Benefit
Supporting synchronised replay of sensory, event and audio data, with replay speed setting	Enabling the testing of applications collecting any of sensory, event and audio data. Allowing tests which are faster or slower than the real-time clock.
Supporting testing data from online, local source and manipulation	Increasing the chance of dataset reuse. Simplifying the data preparation phase before tests.
Supporting black-box testing without requiring any modification for receiving data	Simplifying the process of black-box testing. Encouraging tests performed by testers which do not have corresponding programming skills.
Enabling white-box testing	Allowing testers to examine internal details of application and locate bugs from low levels to high levels.
Providing a code library for white-box testing and non-functional testing	Automating data preparation, white-box testing and non-functional testing. Simplifying non-functional testing. Allowing testers to conduct white-box testing and non-functional testing at the same time.

6.5 Limitations and Future Work

TestAWARE is implemented on Android. We are aware that the fragmentation of Android causes the inconsistency of sensor value representations from different hardware manufacturers. E.g., for the proximity sensor, some manufacturers use {0, 1} to represent a negative or positive. However, other manufacturers use {0, 15} or {0, 100}. These representations are all valid on Android platforms. If sensor values are collected on one device, it may be challenging for another device to recognise the values of the data replay in the testing. Developers may have to transform the representations of the datasets for testing their applications.

Regarding the implementation of TestAWARE on iOS, we found that the sandbox policy of iOS higher than 7.0 does not allow any inter-process communication, meaning that the targeted application can never receive the replayed data from an external testing tool. To perform effective data replaying, a tool must act as a module inside the targeted application (e.g., as a combination of a file reader and data replayer which the targeted application can import). However, this implies the modification of the source code of the targeted application, causing black-box testing to become impossible. Future research can investigate the suitable design of the tool for iOS, as well as other unpopular mobile platforms.

In our experiments, we quantified the maximal replay speed of audio, sensor and event data on smartphones, tablets and PC-based emulators. The experiment used one thread and one type of data per time.

In practice, the testing often involves heterogeneous data in the replay. When replaying multiple data sources at a high speed, the maximal replay speed may reduce because the number of CPU cores may be lower than the number of data sources.

For future work, we plan to include video data in the replay since most smartphones and tablets have dual cameras. Also, smartwatches with diverse sensors are becoming increasingly popular recently. We plan to design a smartwatch version of TestAWARE for the testing of smartwatch-based context-aware applications.

7 CONCLUSION

In this paper, we present TestAWARE, a laboratory-oriented testing tool for mobile context-aware applications, which can download, replay and construct contextual data on either physical devices or emulators. To support both black-box and white-box testing, TestAWARE is designed and implemented as a novel architecture with two components: a mobile client and code library. In black-box testing, developers can manage data replay from the mobile client without writing testing scripts or changing the source code of the targeted application. In white-box testing, the code library supports developers to automate data replay by writing testing scripts and conduct functional examinations using run-time assertions. With the code library, developers can also test non-functional properties of the targeted application, including machine learning performance, energy use and processing speed in real-time clock. We evaluated TestAWARE by quantifying its maximal data replay speed and conducting a user study with 13 professional developers. We found that PC-based emulators can replay data significantly faster than physical devices including smartphones and tablets. The results of the user study confirm the usefulness of TestAWARE in the testing of mobile context-aware applications in the laboratory settings. In the scope of testing mobile context-aware applications, our work provides multiple implications to conduct tests and testing tool design, which can guide both testers and developers (including researchers) of testing tools.

ACKNOWLEDGMENTS

This work is partially funded by SocialNUI, the Academy of Finland (Grants 276786-AWARE, 286386-CPDSS, 285459-iSCIENCE, 304925-CARE), the European Commission (Grants PCIG11-GA-2012-322138, 6AIKA-A71143-AKAI), Marie Skłodowska-Curie Actions (645706-GRAGE) and University of Oulu (Grants ITEE-2016-SA-13, and ITEE-2016-SA-20).

REFERENCES

[1] Domenico Amalfitano, Anna R. Fasolino and Porfirio Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *International Conference on Software Testing, Verification and Validation Workshops*, IEEE, 252-261. http://dx.doi.org/10.1109/ICSTW.2011.77.

[2] Domenico Amalfitano, Anna R. Fasolino, Porfirio Tramontana and Nicola Amatucci. 2013. Considering Context Events in Event-Based Testing of Mobile Applications. In *International Conference on Software Testing, Verification and Validation Workshops*, IEEE, 126-133. http://dx.doi.org/10.1109/ICSTW.2013.22.

[3] Application Exerciser Monkey. Retrieved 17/12/2014 from http://developer.android.com/tools/help/monkey.html

[4] Martin Atzmueller and Katy Hilgenberg. 2013. Towards Capturing Social Interactions with SDCF: An Extensible Framework for Mobile Sensing and Ubiquitous Data Collection. In *Proceedings of the 4th International Workshop on Modeling Social Media*, ACM, 6:1-6:4. http://dx.doi.org/10.1145/2463656.2463662.

[5] Rajesh K. Balan, Darren Gergle, Mahadev Satyanarayanan and James Herbsleb. 2007. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, 272-285.

[6] Szymon Bobek, Sebastian Dziadzio, Paweł Jaciów, Mateusz Ślażyński and Grzegorz J. Nalepa. 2015. *Understanding Context with ContextViewer – Tool for Visualization and Initial Preprocessing of Mobile Sensors Data.* Springer International Publishing.

[7] David Chu, Nicholas D. Lane, Ted T. Lai, Cong Pang, Xiangying Meng, Qing Guo, Fan Li and Feng Zhao. 2011. Balancing Energy, Latency and Accuracy for Mobile Sensor Data Classification. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, ACM, 54-67. http://dx.doi.org/10.1145/2070942.2070949.

[8] Karen Church and Barry Smyth. 2008. Understanding the intent behind mobile information needs. In *Proceedingsc of the 13th international conference on Intelligent user interfaces - IUI '09*, 247-256. http://dx.doi.org/10.1145/1502650.1502686.

[9] Context Simulator (KnowMe). Retrieved 14/04/2016 from http://glados.kis.agh.edu.pl/doku.php?id=pub:software:contextsimulator:start

[10] Denzil Ferreira, Vassilis Kostakos and Anind K. Dey. 2015. AWARE: mobile context instrumentation framework. *Frontiers in ICT* 2, 6: 1-9. http://dx.doi.org/10.3389/fict.2015.00006.

[11] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim and Todd Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *International Conference on Software Engineering*, IEEE, 72-81. http://dx.doi.org/10.1109/ICSE.2013.6606553.

[12] Tobias Griebe and Volker Gruhn. 2014. A model-based approach to test automation for context-aware mobile applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 420-427.

[13] Matthew Halpern, Yuhao Zhu, Ramesh Peri and Vijay J. Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *International Symposium on Performance Analysis of Systems and Software*, IEEE, 215-224. http://dx.doi.org/10.1109/ISPASS.2015.7095807.

[14] Kaasila, Denzil Ferreira, Vassilis Kostakos and Timo Ojala. 2012. Testdroid: automated remote UI testing on Android. In *International Conference on Mobile and Ubiquitous Multimedia*, ACM, 28:1-28:4. http://dx.doi.org/10.1145/2406367.2406402.

[15] Reed Larson and Mihaly Csikszentmihalyi. 1983. The Experience Sampling Method. In *Flow and the Foundations of Positive Psychology* (eds.). Wiley Jossey-Bass, San Francisco, 15, 41-56.

[16] Heng Lu, W. K. Chan and T. H. Tse. 2006. Testing Context-aware Middleware-centric Programs: A Data Flow Approach and an RFID-based Experimentation. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 242-252. http://dx.doi.org/10.1145/1181775.1181805.

[17] Chu Luo, Miikka Kuutila, Simon Klakegg, Denzil Ferreira, Huber Flores, Jorge Goncalves, Vassilis Kostakos and Mika Mäntylä. 2016. How to Validate Mobile Crowdsourcing Design? Leveraging Data Integration in Prototype Testing. In *International Joint Conference on Pervasive and Ubiquitous Computing Adjunct*, ACM. http://dx.doi.org/10.1145/2968219.2968586.

[18] Qi Luo, Denys Poshyvanyk, Aswathy Nair and Mark Grechanik. 2016. FOREPOST: A Tool for Detecting Performance Problems with Feedback-driven Learning Software Testing. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ACM, 593-596. http://dx.doi.org/10.1145/2889160.2889164.

[19] Chulhong Min, Seungchul Lee, Changhun Lee, Youngki Lee, Seungwoo Kang, Seungpyo Choi, Wonjung Kim and Junehwa Song. 2016. PADA: Power-aware Development Assistant for Mobile Sensing Applications. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ACM, 946-957. http://dx.doi.org/10.1145/2971648.2971676.

[20] monkeyrunner | Android Developers. Retrieved 03/05/2016 from http://developer.android.com/tools/help/monkeyrunner_concepts.html

[21] Henry Muccini, Antonio D. Francesco and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*, 29-35.

[22] Zhengrui Qin, Yutao Tang, Ed Novak and Qun Li. 2016. MobiPlay: A Remote Execution Based Record-and-replay Tool for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering*, ACM, 571-582. http://dx.doi.org/10.1145/2884781.2884854.

[23] Kiran K. Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J. Rentfrow, Chris Longworth and Andrius Aucinas. 2010. EmotionSense: a mobile phones based adaptive platform for experimental social psychology research. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, ACM, 281-290. http://dx.doi.org/10.1145/1864349.1864393.

[24] Luis Roalter, Andreas Moller, Stefan Diewald and Matthias Kranz. 2011. Developing intelligent environments: A development tool chain for creation, testing and simulation of smart and intelligent environments. In *Intelligent Environments (IE), 2011 7th International Conference on*, 214-221.

[25] M Satyanarayanan. 2001. Pervasive computing: Vision and challenges. *Personal Communications, IEEE* 8, 4: 10-17. http://dx.doi.org/10.1109/98.943998.

[26] Choonsung Shin, Jin-Hyuk Hong and Anind K. Dey. 2012. Understanding and Prediction of Mobile Application Usage for Smart Phones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, ACM, 173-182. http://dx.doi.org/10.1145/2370216.2370243.

[27] Melanie Swan. 2013. The Quantified Self: Fundamental Disruption in Big Data Science and Biological Discovery. *Big Data* 1, 2: 85-99. http://dx.doi.org/10.1089/big.2012.0002.

[28] Ralf Tonjes, Eike S. Reetz, Marten Fischer and Daniel Kuemper. 2015. Automated Testing of Context-Aware Applications. In *Vehicular Technology Conference*, IEEE, 1-5. http://dx.doi.org/10.1109/VTCFall.2015.7390847.

[29] Niels van Berkel, Chu Luo, Theodoros Anagnostopoulos, Denzil Ferreira, Jorge Goncalves, Simo Hosio and Vassilis Kostakos. 2016. A Systematic Assessment of Smartphone Usage Gaps. In *Conference on Human Factors in Computing Systems*, ACM, 4711-4721. http://dx.doi.org/10.1145/2858036.2858348.

[30] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas and Denys Poshyvanyk. 2015. Generating Reproducible and Replayable Bug Reports from Android Application Crashes. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, IEEE Press, 48-59.

Received November 2016; revised May 2017; accepted July 2017