A Survey of Context Simulation for Testing Mobile Context-Aware Applications

CHU LUO, JORGE GONCALVES, EDUARDO VELLOSO, and VASSILIS KOSTAKOS,
University of Melbourne

Equipped with an abundance of small-scale microelectromechanical sensors, modern mobile devices such as smartphones and smartwatches can now offer context-aware services to users in mobile environments. Although advances in mobile context-aware applications have made our everyday environments increasingly intelligent, these applications are prone to bugs that are highly difficult to reproduce and repair. Compared to conventional computer software, mobile context-aware applications often have more complex structures to process a wide variety of dynamic context data in specific scenarios. Accordingly, researchers have proposed diverse context simulation techniques to enable low-cost and effective tests instead of conducting costly and time-consuming real-world experiments. This article aims to give a comprehensive overview of the state-of-the-art context simulation methods for testing mobile context-aware applications. In particular, this article highlights the technical distinctions and commonalities in previous research conducted across multiple disciplines, particularly at the intersection of software testing, ubiquitous computing, and mobile computing. This article also discusses how each method can be implemented and deployed by testing tool developers and mobile application testers. Finally, this article identifies several unexplored issues and directions for further advancements in this field.

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

Additional Key Words and Phrases: Mobile devices, sensors, software testing, multimedia

ACM Reference format:
Chu Luo, Jorge Goncalves, Eduardo Velloso, and Vassilis Kostakos. 2020. A Survey of Context Simulation for Testing Mobile Context-Aware Applications. *ACM Comput. Surv.* 53, 1, Article 21 (February 2020), 39 pages.
https://doi.org/10.1145/3372788

1 INTRODUCTION

With the prevalence of small-scale microelectromechanical sensors, modern mobile devices such as smartphones and smartwatches can now provide context-aware services to users in the mobile environment. Mobile context-aware applications are increasingly emerging in multiple domains.

This work was partially funded by a Samsung Global Research Outreach grant, the ARC Discovery Project DP190102627, and Google. E. Velloso was the recipient of an Australian Research Council Discovery Early Career Award (Project Number: DE180100315).
Authors' address: C. Luo, J. Goncalves, E. Velloso, and V. Kostakos, School of Computing and Information Systems, University of Melbourne, Parkville, VIC, 3010, Australia; emails: {chu.luo, jorge.goncalves, eduardo.velloso, vassilis.kostakos}@unimelb.edu.au.

0360-0300/2020/02-ART21 $15.00
https://doi.org/10.1145/3372788

A renowned example is Uber [105], a location-aware smartphone application that helps passengers hire regular vehicles driven by casual drivers. On the user interface (UI) of Uber, passengers can see their own location and that of nearby vacant vehicles. Moreover, mobile context-aware applications based on advanced machine learning algorithms can recognise complex context from sensor data to provide relevant services. For example, CarSafe [117] is a smartphone application enabling safe driving by monitoring the driver behaviour and road conditions. From a smartphone mounted on the car windscreen, it uses the front camera to monitor the driver and the rear camera to track road conditions. It also uses the Global Navigation Satellite System (GNSS) (e.g., Global Position System), accelerometer, and gyroscope to infer the vehicle's movement. If dangerous driving behaviours are detected, CarSafe will instantly trigger alerts.

Although advances in mobile context-aware applications have made our everyday environments increasingly intelligent, these applications are prone to bugs that are highly difficult to reproduce and repair [33, 75, 102]. Compared to conventional computer software, mobile context-aware applications often have more complex structures to process a wide variety of dynamic context data in real time. Traditional software testing techniques cannot effectively examine the behaviours of such applications corresponding to dynamic environments. In addition, as mobile context-aware applications are generally designed for specific scenarios (e.g., particular locations and user activities), it is often laborious, time consuming. and costly to conduct real-world tests [69, 102].

Consequently, researchers have proposed multiple methods that can use simulated context data to test mobile context-aware applications. However, designing context simulation methods for mobile context-aware applications is a highly challenging task for three reasons:

(1) *Platform limitations*: In Integrated Development Environments (IDEs) of popular mobile platforms, it is easy to simulate UI events such as clicks and touchscreen gestures on emulators. In addition, it is possible to simulate a handful of context events on emulators via manual input in IDEs such as Android Studio [38] and Xcode [7]. Some IDEs can offload such simulation to cloud-based testing platforms. Firebase Test Lab [40] and Xamarin Test Cloud [77] host various physical devices and emulators on the cloud to simulate manual input and several system events for Android and iOS applications. However, on current IDEs and cloud platforms, there is little support for advanced context simulation regarding more context event types.
(2) *Context heterogeneity*: Typically, mobile context-aware applications rely on various sensor types. Due to differences in low-level design, each sensor type may need a unique mechanism that is suitable for simulation in practice.
(3) *Ecosystem fragmentation*: The architectures and features of mobile platforms are often different. Although certain actions are supported on one platform, they may not work on another. Researchers often struggle for platform-specific design of context simulation.

Although technically challenging, context simulation is a popular way for testers to avoid laborious, time-consuming, and costly real-world tests. Here, we provide a comprehensive overview of state-of-the-art context simulation methods for testing mobile context-aware applications.

1.1 Related Work

The literature has several survey and tutorial articles on mobile application testing [19, 29, 58, 65, 99, 103, 121], as summarised in Table 1.

Gao et al. [29] present a tutorial of mobile application testing in terms of (1) requirements, (2) activities, (3) approaches, (4) environments, and (5) automation. For each area, they provide a high-level comparison among major features of 15 popular mobile testing tools.

Table 1. Summary of Related Work

Study	Focus	Number of Studied Works	Year of Publication
Gao et al. [29]	Features of generic mobile testing tools	15	2014
Choudhary et al. [19]	Features of Android GUI testing tools	14	2015
Starov et al. [99]	Cloud-based mobile application testing	34	2015
Zein et al. [121]	Systematic mapping on mobile testing research	79	2016
Linares-Vásquez et al. [65]	Technical features of automated mobile testing	81	2017
Kong et al. [58]	Automated testing techniques for Android apps	103	2018
Tramontana et al. [103]	Systematic mapping on automated functional mobile testing	131	2018
This study	Context simulation techniques for testing mobile context-aware apps	51	—

Choudhary et al. [19] compare and evaluate several Android testing tools that have automatic input generation. As the focus is mainly GUI testing, most of evaluated tools can only generate GUI events. Little is discussed about system events such as sensor readings and hardware states.

Starov et al. [99] make a brief review of cloud-based services and stand-alone tools for mobile application testing. They point out that cloud-based testing environments for mobile applications can serve as testing-as-a-service (TaaS) resources which help developers, especially startup companies, conduct low-cost and efficient tests.

Zein et al. [121] conduct a systematic mapping study on empirical research in the domain of mobile application testing. They compare technicalities, contributions, object software applications (e.g., real-world applications, sample applications), and research methodologies by classifying testing approaches into five high-level categories: test automation, usability, context-awareness, security, and general testing techniques.

Linares-Vásquez et al. [65] provide a brief review of automated testing techniques for mobile applications. They discuss the main technical features of automated mobile testing approaches in seven categories: automation frameworks, record-and-replay tools, automated test input generation, bug tracking tools, testing services, cloud-based testing services, and device streaming tools.

Kong et al. [58] summarise the automated testing techniques for Android applications regarding test objectives (e.g., security, performance), test targets (e.g., events, communication), test levels (e.g., unit testing, system testing), and test techniques. Beyond general challenges of mobile testing, their survey also covers Android-specific aspects, such as the fragmentation problem.

Tramontana et al. [103] perform a systematic mapping study on automated functional testing techniques for mobile applications. They focus on testing activities of functional testing, characteristics of testing techniques, and evaluation methodologies used in research experiments.

The literature mainly has focused on two kinds of topics: (1) generic features of mobile application testing and (2) specialised tools for mobile application testing. These specialised tools may only work for limited cases. For example, Android GUI testing tools cannot be used for iOS applications. In addition, some previous studies only aimed to discuss the features of mobile testing tools, without providing insights and deliberations about technical mechanisms behind. Although the changing context of mobile devices and applications is often mentioned in the literature, little attention has been paid to building an overview of the state-of-the-art context simulation techniques. A detailed review in this field is needed but lacking.

Table 2. Built-In Sensors on Common Smartphones, Smartwatches, and Tablets

		Device		
		Smartphone	Smartwatch	Tablet
Sensor	Motion	Accelerometer Gyroscope	Accelerometer Gyroscope	Accelerometer Gyroscope
	Environment	Ambient Temperature Barometer, Light Device Temperature Relative Humidity	Ambient Temperature Barometer, Light Relative Humidity	Light
	Position	Cellular Antenna Magnetometer GNSS, Proximity	GNSS, Magnetometer Proximity	GNSS Magnetometer
	Multimedia	Camera, Microphone	Microphone	Camera Microphone
	Connectivity	Bluetooth, NFC, USB WiFi	Bluetooth, NFC, USB WiFi	Bluetooth NFC, USB WiFi
	Health	PPG	PPG, ECG	

2 BACKGROUND

2.1 Mobile Devices and Sensors

After outnumbering personal computers (e.g., desktops) in 2013, mobile devices have become the most popular computers. People are increasingly linked to the digital world via mobile devices such as smartphones, smartwatches, tablets, and wearable computers. In developed countries, smartphones are adopted by more than 90% of the population. Now we are witnessing a quickly shrinking gap by developing nations. In addition, smartwatches, tablets, and wearable computers are gradually gaining popularity. Given that mobile context-aware applications generally operate on smartphones, tablets, and smartwatches, we focus our analysis on the scope of these three device types.

2.1.1 Basic Modules. Mobile devices have several essential components that a regular computer also has, including a Central Processing Unit (CPU), memory, power supply (i.e., battery on mobile devices), screen, and buttons/keys. Although these modules provide considerable computational capabilities, resources on mobile devices remain more constrained compared to non-mobile devices. A notorious example is their short battery life. Similarly, due to the small screen size and virtuality of keyboards, text input on smartphones is inefficient, and even more so on smartwatches. Beyond supporting computational purposes, these essential components, including the CPU, battery, and screen, can also provide information (e.g., CPU workload) as useful clues for context recognition.

2.1.2 Built-In Sensors. Modern mobile devices come with multiple built-in sensors. Table 2 classifies them into six classes: motion, environment, position, multimedia, connectivity, and health.

Motion sensors (i.e., accelerometers and gyroscopes) are low-power and small size microelectronic instruments that provide sensitive acceleration and rotation measurements. When the user carries the device, the accelerometer provides high-quality information indicating the physical activity.

Environmental sensors measure a variety of physical quantities: ambient/device temperature, luminosity, atmospheric pressure, and relative humidity. Due to their diversity, infrequent usage, and limited space of each device, manufacturers may embed only some of them into a certain device model. Environmental sensors collect information from the world regardless of user presence. Thus, they can sense both user activities and phenomena of environments.

Position sensors provide information to locate a mobile device. The proximity sensor works by detecting whether there is a physical object near the device surface. A typical usage scenario of proximity sensor is to detect whether the phone is in a pocket. Magnetometers measure magnetism which is strongly influenced by the position of the device. Since the geomagnetic field exists and varies in any place on the earth's surface, a magnetometer serves as a compass in outdoor environments. In indoor environments, the magnetic field also varies within buildings, enabling magnetometer-based indoor localisation approaches. In contrast, GNSS directly offers latitude and longitude coordinates to the device. However, indoor environments may impede the GNSS signal. Another solution for explicit location coordinates is the cellular network that locates devices by estimating the distance and/or angles to nearby cellular towers.

Multimedia components (e.g., the camera and microphone) are not primarily sensors, but recent machine learning approaches are able to recognise context from audio and video. Most smartphones and tablets have dual cameras (front and back) which can simultaneously record video from two directions. Several approaches infer context using machine vision algorithms and dual cameras, such as identifying driving behaviour on the road. However, the camera may be blocked in many cases (e.g., devices are in pockets or bags). Comparatively, the microphone has fewer constraints to record audio for context recognition, leading to mobile sound sensing systems that can infer human activities and ambient events.

Connectivity transceivers include Bluetooth, which enables short-range device-to-device detection and communication. It is a crucial component for most smartwatches to access information (e.g., app installation, Internet, and locations) via smartphones, because not many smartwatches have WiFi, cellular, or GNSS modules. Since Bluetooth can find nearby devices without establishing a paired connection, researchers have built various Bluetooth-based systems on mobile devices to infer both social and environmental context [27]. WiFi (i.e., IEEE 802.11) is another wireless technology that allows devices to connect to networks such as the local area network (LAN) and the Internet. Similar to Bluetooth, WiFi is also used for social context recognition [27]. In addition, researchers have proposed WiFi localisation approaches (e.g., Liu et al. [66]) using Receive Signal Strength (RSS) readings from an array of WiFi access points. Near-Field Communication (NFC) offers contactless communication within a close distance, enabling use cases such as contactless payment systems. NFC-based mobile devices can also trigger smart services by reading NFC tags, such as home care systems. In addition, a Universal Serial Bus (USB) is an essential component of mobile devices for the purpose of recharging. Mobile devices can also use USB to receive data from integrative pluggable sensors.

Health sensors on mobile devices are currently photoplethysmogram (PPG) and electrocardiography (ECG) sensors. PPG can monitor the human heart rate and blood oxygen level (SpO2) by tracking the light absorption of an illuminated finger. ECG, mainly on smartwatches, monitors cardiac electrical potential waveforms from heartbeats.

2.1.3 Software Applications and Human Input. A wide variety of software applications are designed for mobile devices, including browsers, email, music, game, and social media, just to name a few. They generate heterogeneous data which contains valuable insights of the user's context for personal and contextual services. For example, using the data generated by software applications, mobile devices can infer one's health status, such as mental health.

In some cases, humans can describe their current context much more accurately than hardware or software, such as when it comes to measuring physical pain. Based on simple experience sampling methods such as phone diary and surveys, mobile users can directly input descriptions that are relevant to the goals of context-aware services. Especially tablets, due to their bigger size, can serve as an essential tool for human input in workplace environments.

2.1.4 External Data Sources. Mobile devices can obtain data from sources other than their built-in sensors, such as USB pluggable sensors. One such major category is the biochemical sensor. Researchers have integrated a wide range of biosensor technology into smartphone sensing systems [24], including the colorimetric test strip, electrochemical detection, microfluidic genetic analysis, and microscopy. With such an integration, mobile users can benefit from additional services, ranging from nutrition and micronutrient management to disease diagnostics. Similarly, the combination of additional portable electronic sensors and mobile devices also creates numerous opportunities in mobile context-aware computing.

However, smart objects and stationary sensors in the Internet of Things (IoT) can exchange information with mobile devices via diverse heterogeneous communication channels. Thus, mobile devices are able to improve context recognition using environmental phenomena and physical events captured by IoT. A typical scenario is intelligent transportation where mobile devices guide car drivers to avoid traffic jam using road monitoring systems in IoT.

2.2 Mobile Context-Aware Applications

Enabled by the data richness that the preceding sources provide, researchers and developers have built diverse mobile context-aware applications for a multitude of purposes. In this section, we discuss the general characteristics of mobile context-aware applications.

2.2.1 Architecture. To describe the architecture of conventional desktop-based and server-based context-aware systems, an early work [9] proposed a layered conceptual framework consisting of the following:

(1) *Sensors*: Sensors represent not only hardware sensors but all data sources providing clues for context recognition, including physical sensors (i.e., hardware), virtual sensors (i.e., software applications, calendar) and logical sensors (i.e., combinations of data sources).
(2) *Raw data retrieval*: Retrieval of raw context data involves the invocation of drivers and APIs connected to data sources. Details such as the sampling rate of sensors are implemented.
(3) *Preprocessing*: The preprocessing phase refines coarse-grained raw data into descriptive information. For example, GNSS coordinates are converted to specific locations.
(4) *Storage/management*: The layer of storage and management gathers and redistributes all of the data, normally using an information system.
(5) *Application*: The application layer includes information retrieval, context reasoning, and services to end users. These actions are encapsulated on client devices.

Since this context-aware framework was not primarily designed for mobile devices, its architecture contains the process of information management and exchange with the help of online systems. This outdated framework does not fit the current view on the context processing mechanism of mobile applications. Comparatively, modern mobile devices are able to locally collect, store, and process data. A subsequent work on mobile phone sensing [59] suggests a simpler architecture:

(1) *Sense*: The sense phase refers to not only raw data collection from sensors but also the scheduling strategies for energy-efficient continuous sensing.
(2) *Learn*: The learning phase interprets raw data to gain useful knowledge. Features and patterns are extracted from raw data by context models and algorithms, such as machine learning.
(3) *Inform, share, and persuasion*: The last phase of mobile phone sensing systems is *inform, share, and persuasion*, where human-readable information is presented to end users. For specific goals, users may receive such feedback in varying modalities, such as using a chronological achievement graph to spur users in weight loss.

Considering that some mobile context-aware systems are rule based, the learning phase may not be needed. Due to the similarity of data management between smartphones and other mobile devices, the architecture of mobile context-aware applications should conform to that of mobile phone sensing applications. In addition, some mobile context-aware systems are designed to take action rather than interact with users, such as temperature controls of smart home systems. Biegel and Cahill [10] use the term *actuator* to describe the process after context recognition.

2.2.2 Examples of Mobile Context-Aware Applications. Mobile context-aware applications may use one or more sensors, are able to analyse the collected information, and provide services locally on mobile devices. Several representative examples include the following:

(1) SoundSense [68] is a smartphone-based application that can infer the ambient sound events in users' daily life. It collects only audio data from the microphone on smartphones. Then, it splits raw data into short frames and extracts features in temporal and spectral domains. Using these features, it classifies sound events into {*speech, music, ambient sound*} by a decision tree classifier and Markov models. Within these three sound event classes, it can further generate finer classification, such as music genre.
(2) CarSafe [117] is smartphone application which aims to ensure safe driving by monitoring the driver behaviour and road conditions. On a smartphone mounted on the car windscreen, it uses a front camera to monitor the driver states and a rear camera to track road conditions. Beyond the visual information of dual cameras, it also relies on the GNSS, accelerometer, and gyroscope to infer the vehicle's movement. If dangerous driving behaviour is detected, CarSafe triggers alerts.

2.2.3 Summary. The diversity of sensors makes mobile devices a kind of multipurpose platform. By collecting contextual information in physical and social environments, mobile context-aware applications can infer various situations of users and deliver more relevant services. As people frequently use their mobile devices in most places, mobile context-aware applications are designed for an increasingly wide range of purposes in daily life.

2.3 The Need of Context Simulation for Testing Mobile Context-Aware Applications

As mobile context-aware applications respond to certain contexts, testing these applications requires contextual input generated in either simulated conditions or real-world scenarios [83]. Although real-world tests provide realistic situations for mobile context-aware applications to operate, testers may have to bear high cost (e.g., time and money) [83]. Moreover, the dynamic nature of real-world environments, and random hardware drifts, can influence the reproducibility of test results.

Hence, testers often conduct context simulation to examine mobile context-aware applications for lower testing cost, higher automation, and guaranteed result reproducibility [33, 87]. To simulate context, testers have two options [4]:

(1) *Data-driven approaches*: Data-driven context simulation aims to directly test actual software applications by delivering simulated sensor data. Testers can deliver test data by writing scripts or using a data replay/generation tool. Such sensor data drives context recognition modules of mobile context-aware applications so that the response of applications can be triggered and analysed.
(2) *Model-based approaches*: Model-based context simulation mainly focuses on the examination of software design models, whereas other model-based approaches generate test cases to examine actual software. Testers usually have to create a model as an abstraction of software (sometimes also including the intended environment) for simulation tools to analyse. Most model-based context simulation tools can automatically generate abstract test cases that can change the states of a model. The transitions of states can represent the designed dynamic behaviours of target systems.

For various domain-specific contexts, context simulation via either data-driven or model-based approaches is a multifaceted process involving the following phases:

(1) *Test case acquisition*: For most data-driven approaches, testers should prepare sensor data as concrete test cases before simulation. During simulation, such data is used to reconstruct the intended context where applications operate. In addition, a small number of model-based approaches require testers to provide abstract test cases, although most model-based approaches can automatically generate abstract test cases to conduct simulation.
(2) *Test case refinement*: Based on initial test cases, some context simulation approaches can further generate a set of filtered or augmented test cases. These refined test cases are more likely to achieve higher coverage or efficiency.
(3) *Test execution*: Context simulation approaches execute test cases on target models or software applications to examine different properties. Data-driven approaches operate by delivering raw sensor data to mobile context-aware applications which may run on physical devices, emulators, servers, or the cloud. Comparatively, most model-based approaches use high-level states and actions to validate the design of models, although a few model-based approaches also generate concrete test cases for execution on actual applications.

3 DATA-DRIVEN CONTEXT SIMULATION

In this section, we summarise data-driven context simulation approaches, whereas in the next section, we turn to model-based context simulation. In this section, we consider the three phases we have identified: acquisition (Section 3.1), refinement (Section 3.2), and execution (Section 3.3).

3.1 Test Data Acquisition

3.1.1 Record-and-Replay. Record-and-replay approaches play a crucial role in software testing for their ability to automate test data acquisition and test execution. With a record-and-replay tool capturing data at runtime, testers can easily obtain test data by interacting with the targeted application as usual. Then, testers can use the same tool to replay the captured data to simulate context for testing purposes. Before the emergence of mobile applications, record-and-replay techniques were widely used for testing PC and server applications. For example, Jovic et al. [56] present a Java GUI test automation tool that can capture and replay users' mouse and keyboard events on a PC. To capture contextual data for testing mobile context-aware applications, researchers and developers have proposed various record-and-replay tools for mobile platforms, as summarised in Table 3.

Table 3. Data-Driven Context Simulation Approaches Relying on Data Recording

Approach	Recording Mechanism	Supported Data Type
RERAN [33]	*getevent* [36]	GUI action, system event, sensor
MobiPlay [87]	*getevent*	GUI action, system event, sensor
	API invocation	Location
VALERA [49, 51]	Event interception	GUI action, network, image, audio, IPC
	API invocation	Sensor, location, random number, I/O
ODBR [81]	*getevent*	GUI action
	API invocation	Sensor, location
MoTiF [34]	Event interception	GUI action
	API invocation	system event, sensor, network, exception
DroidForensics [119]	Event interception	API use, IPC, system call
Paranoid [85]	Event interception	System event, sensor, network, exception

Gomez et al. [33] presented an Android-based record-and-replay tool called *RERAN*. RERAN can record three types of low-level events with timestamps: touchscreen input events, physical sensor data, and OS events. The supported physical sensors include an accelerometer, proximity, and a magnetometer. However, RERAN cannot capture GNSS or network location information because Android encapsulates localisation as a specialised service. The recording feature of RERAN relies on Android's *getevent* tool which runs inside the Linux kernel of Android. *getevent* can be called manually via the command line or programmatically by code through the Android Debug Bridge (ADB). The captured events and data are saved in an event trace file that can be replayed.

Regarding the limited event types supported by RERAN, Qin et al. [87] designed a record-and-replay tool called *MobiPlay* which can capture a wider variety of events on Android devices. Its supported event types to capture consist of touchscreen events, key events, physical sensor data, screen orientation, and location information. For location information recording, MobiPlay uses Android *LocationListener* API which operates as a specialised service and reports the location change, location provider (GNSS or network), and the state of location providers (enabled/disabled). MobiPlay can also track the angle of screen orientation within {0, 90, 180, 270} degrees. MobiPlay stores each captured event as a structure called *Request*. Representing all of the captured events, the set of *Request* objects are written into an *InputLog* instance on the device disk for replay. Since the formatted *Request* objects are human readable and revisable, testers can use them not only for regular context simulation but also data analysis or data augmentation.

Furthermore, Hu et al. [49] and Hu and Neamtiu [51] proposed a versatile record-and-replay tool for Android called *VALERA*. Beyond GUI events and physical sensor data, VALERA can record the events and data generated by a camera and microphone. In addition, VALERA captures the network state, traffic, exceptions, and API calls of HTTP/HTTPS connections. Since mobile context-aware applications may involve randomised algorithms (they make random decisions in the same context), VALERA records the seeds of random number generators (two Android APIs: *java.util.Random* and *java.security.SecureRandom*) at runtime. In addition, VALERA captures Android Intent events which are widely used in inter-application communications to transfer parameters and/or to launch new software components.

Unlike general-purpose record-and-replay tools, Moran et al. [81] developed a bug-oriented record-and-replay prototype called *ODBR* (completed as CrashScope [82]). ODBR focuses on the functionality of bug reporting so that testers can find the correlations between bugs and captured contextual information. With the knowledge of such correlations, testers can more efficiently fix the bugs. ODBR can capture GUI input events, sensor data, system events, and screenshots to

Table 4. Data-Driven Context Simulation Approaches Relying on External Data Sources

Approach	Data Format	Supported Data Type
Hermes [95]	XML	GUI action, network
PUMA [44]	PUMAScript	GUI action, system event
Node.fz [22]	Node.js	System event
Thor [2]	Format neutral	GUI action, system event, sensor, network
Amalfitano et al. [4]	JUnit	GUI action, system event, network
MobileTest [12]	Database	GUI action, system event, sensor, network
Caiipa [62]	Database	GUI action, system event, sensor, network
KnowMe [13]	AWARE [25]	System event, sensor, network
TestAWARE [70]	AWARE, WAV	System event, sensor, network, audio
Hu and Neamtiu [50]	VALERA [49, 51]	Location, image, audio
CamTest [71]	MP4 video	Single-camera video

describe a bug in a JSON file. This JSON file is saved on the cloud where a web application generates a html report for testers to read. Using this web application, testers can also download an executable script to replay the recorded contextual data on a physical device for bug reproduction.

Another bug-oriented record-and-replay tool called *MoTiF* [34] collects contextual data and app crash information using a crowdsourced approach. MoTiF has a mobile app client operating on devices of a crowd of users. This client captures the data when users interact with their devices "in the wild." Once an app crashes, the client will send the captured data to the MoTiF server at the next moment when the device is charging (to preserve battery life) and linked to the Internet. The MoTiF client captures three types of data: GUI event metadata (including user input and device response), exception metadata (i.e., crash information), and context data.

In addition, a specialised record-and-replay tool, called *DroidForensics*, in Yuan et al. [119] aims to reconstruct cyber attacks on Android devices for post-mortem analysis. DroidForensics records API calls onto Android framework resources (e.g., fetching installed application list), hardware modules (e.g., accessing a microphone, camera, or GNSS), and storage (e.g., database or Android's *ContentProvider* usage). In addition, DroidForensics captures Inter-Process Communication (IPC) between different applications. DroidForensics can also detect system calls (e.g., the usage of native components) using a module inside Android's Linux kernel. DroidForensics assigns each event a timestamp in its log so that the replay process can reproduce the original sequence of events.

Similarly, Portokalidis et al. [85] proposed a record-and-replay tool called *Paranoid Android* for malware detection on Android smartphones. Paranoid Android has a data collection module called *tracer* which can record inputs and events of an application using the process identifier (pid) of the targeted process. *tracer* captures system calls containing all hardware messages (e.g., device clock, GNSS location, sensor data, and network traffic) sent from Linux kernel to user space. In addition, it records synchronous and asynchronous notifications (e.g., expired timers and runtime exceptions).

3.1.2 External Data Sources. Many data-driven context simulation tools rely on external sources or tools to acquire test data. These context simulation tools allow testers to import test data in specific formats (e.g., SQL databases and XML files). Since testers can use data collected by a third party, these tools can avoid the cost of recording contextual data in real-world scenarios. Table 4 outlines data-driven context simulation approaches relying on external data sources.

A popular way of importing external data is to embed it into test scripts. For example, She et al. [95] presented Hermes, an automated testing framework for mobile applications. On Hermes,

test cases containing contextual data are represented in XML schemas. Different GUI actions and network states of mobile devices can be defined as XML elements inside these schemas. Testers can create such XML schemas either manually or by using a test case generator. Similarly, PUMA [44] also imports test cases from test scripts in its own scripting language called *PUMAScript*. Using PUMAScript, testers can define GUI actions and app-specific events (e.g., change of network states).

Node.fz [22] is a fuzz testing tool for Node.js to detect concurrency bugs (e.g., race conditions) in non-deterministic processes. It can fuzz the order of input events, task executions, and completions in the worker pool. Hence, with a small number of input events, Node.fz can explore all of the possible consequences to increase the test coverage. In the same way, Node.fz can reliably reproduce bugs corresponding to specific input events. Furthermore, some language-neutral methodologies allow testers to inject contextual events into manually written test scripts. For example, a lightweight methodology is proposed in Adamsen et al. [2], aiming to support the injection of GUI actions, system events, sensor data, and network states.

Beyond test scripts, Amalfitano et al. [4] designed a context simulation approach that can import test data from external repositories. It supports the simulation of GUI events, system events, and sensor data. Specifically, testers can import GUI events collected by a GUI ripper described in Amalfitano et al. [5]. In addition, testers can write scripts to manually define events to simulate the intended scenarios.

Bo et al. [12] proposed an automated black-box testing tool called *MobileTest* which considers test cases as part of test resources. MobileTest has a database to store all of the test resources, such as configuration information, test plans, test cases, and results. For each kind of test resources, MobileTest provides a unique interface to access data so that the tester can easily manage test resources and analyse results corresponding to certain test cases.

Liang et al. [62] presented an automated large-scale cloud-based testing platform called *Caiipa*, aiming to provide contextual fuzzing [61]. Caiipa consists of various virtual and real smartphones and tablets for testers to conduct remote simulation. Caiipa supports three types of contextual data:

(1) *Wireless network conditions*: Delay, jitter, loss, throughput, and power consumption
(2) *Sensor data and states*: Availability/settings of sensors and sensor readings
(3) *Device specifications*: Chipset, memory size, processors, screen size/resolution, and availability of hardware modules (e.g., NFC and camera).

Caiipa contains a context management module called *ContextLib* which stores a variety of third-party contextual data. Part of the data is collected from real devices, including OpenSignal [96] (crowdsourced observations of mobile networks) and WER [32] (automated error reports from Microsoft applications, Microsoft OS since Windows XP, Windows Mobile, and Windows Live JavaScript programs for browsers). In addition, the data contains test cases specialised by domain experts for particular contexts and device categories.

KnowMe [13] is a Java-based context replay tool that can import data from the third-party data collection framework AWARE [25]. Mobile devices can use the AWARE client to collect data generated by hardware sensors, OS, and human input. The AWARE client sends data to an AWARE server periodically. KnowMe can directly download data from an AWARE server using its API. After downloading the data, KnowMe can replay the data for context simulation on a PC. Based on KnowMe, Bobek et al. [14] proposed a context analysis tool called *ContextViewer* for testers to visualise and to preprocess mobile sensor data. However, ContextViewer cannot perform context simulation for mobile applications. Similarly, using AWARE as the data source, Luo et al. [70] designed a context simulation tool called *TestAWARE*.

Table 5. Data-Driven Context Simulation Approaches Using Self-Generated Data

Approach	Generation Mechanism	Supported Data Type
Majchrzak and Schulte [73]	Random generation	Hardware/software state, system event, sensor, network
Liu et al. [67]	ART	GUI action, system event, sensor
RainDrops [123]	Sampling from physical devices	System event, sensor, network
CATLES [30]	Sampling from physical devices	WiFi, cellular signal
VanarSena [89]	Random generation	GUI action, sensor, network
Song et al. [98]	Analysing declared permissions	Hardware state, network
Usman et al. [106]	Analysing declared permissions	Hardware state, network
	Analysing source code	GUI action, system event
ATT [76]	Analysing declared permissions	Hardware state, IPC
	Analysing source code	GUI action, system event, sensor
Snowdrop [122]	Analysing source code	IPC
Chizpurfle [53]	Fuzz operators	Parameters of method
QGJ [115]	Fuzz operators	GUI action, IPC
ANDROFLEET [74]	Sampling from emulators	WiFi P2P action

TestAWARE aims to conduct context simulation to test Android applications on emulators or physical devices. With TestAWARE, testers can download mobile sensor data from an AWARE server using either API in scripts or the UI of the TestAWARE client. In addition to mobile sensor data, TestAWARE supports the context simulation of audio data to test microphone-based mobile context-aware applications. To import audio data, testers need to transfer a file in Waveform Audio File Format (WAV) on the storage of emulators or physical devices.

Hu and Neamtiu [50] presented a fuzzy and cross-app context replay approach which reads data collected by VALERA. This approach supports context simulation of location, images, and audio with semantic sensor data alteration (SSDA). Moreover, to simulate single-camera video for camera-based applications, CamTest [71] can read and replay video frames from MP4 video files on the Android platform.

3.1.3 Self-Generated Data. Despite record-and-replay tools and external data sources, testers may still need more data to increase test coverage. In addition, recording contextual data in real-world scenarios or using third-party data may involve high cost. Hence, researchers and developers have designed several context simulation tools that can automatically generate test data to test mobile context-aware applications. Table 5 compares data-driven context simulation approaches that can automatically generate test data.

Majchrzak and Schulte [73] proposed a concept of block-based context-sensitive testing. This concept performs gray-box testing by splitting test code into blocks containing assertions. Between blocks, a test case can change the context so that assertions can examine context-dependent behaviours. Generators can be implemented to automatically create test cases. The supported context types include hardware/software states, system events, sensor data, and network states.

From a black-box view, Liu et al. [67] proposed an adaptive random testing (ART) approach that can fully automatically generate test cases without access to the application source code. Unlike common random testing, ART [16] generates random test cases that are more evenly distributed in the input space. For mobile applications, Liu et al. [67] defined a new measure of test case distance by considering sequence and value distance between two lists of events. To measure sequence distance, they employed a common string distance metric, Levenshtein distance [60], which represents the minimal operation to transform a string to another. For value distance, they

transform two event lists into identical types and lengths, and sum up distances between each pair of the events. The distance between two events depends on the data type (e.g., Euclid distance for integers).

Zhang et al. [123] developed a cloud-based testing platform called *RainDrops* for mobile applications. RainDrops aims to examine the stability of mobile applications under various network environments, sensor events, and geo-locations. It also considers device heterogeneity by allocating suitable emulators and devices according to different test specifications. RainDrops offloads the collection of contextual data from emulators to physical devices in the wild. For each test session, RainDrops continuously fetches contextual data generated by physical devices.

In addition, for location-based mobile context-aware applications, CATLES [30] simulates location-related context by replaying crowdsensed WiFi and cellular signals that are publicly available. CATLES provides a 3D virtual view of the world for testers to define a position trace.

Similarly, Ravindranath et al. [89] designed another cloud-based testing platform, VanarSena, for testing Windows Phone apps. VanarSena accepts app binary files from testers. During testing, VanarSena first instruments the target app with additional modules for emulators to inject input and induce faults. When the app is running, VanarSena can emulate contextual data such as HTTP response messages (e.g., 404 Not Found), network conditions (e.g., disconnection), sensor values (e.g., null and extreme values), and GUI actions (e.g., invalid entry and interruption of interaction). In VanarSena, the interception and manipulation of HTTP requests rely on Fiddler [20]. In addition, varying network conditions are simulated using a customised tool similar to Dummynet [91].

To test Android context-aware applications, Song et al. [98] proposed an approach that generates test cases by referring to the permissions requested by the applications. Each Android application contains an *AndroidManifest.xml* file [37] to describe the required permissions of resources, such as the use of WiFi, location, and camera. Users can see the permission list when they attempt to install an application. During tests, this approach generates and explores all possible combinations of states regarding each resource specified in the permission list. Moreover, Usman et al. [106] pointed out that the *AndroidManifest.xml* file cannot provide information about applications' behaviours corresponding to intra-/inter-application communications or events from the OS. Hence, they extend the work of Song et al. [98] by analysing both the *AndroidManifest.xml* file and application source code. The generated test cases can contain hardware-related events, system events, and GUI events. Similarly, another Android testing platform, ATT [76], generates test cases by investigating GUI layout files, the *AndroidManifest.xml* file, and the invocations of system services in source code.

To test background services of Android applications, Zhang et al. [122] developed Snowdrop. It can identify and test background components from source code. Using a code miner, it localises the code block of each background service to parse the required data for service execution. After the analysis of each service, Snowdrop generates test cases that consist of service trigger and execution input. A service trigger input is the unique name of a background service. A service execution input comprises the schemes and values of inbound and outbound messages, such as application local storage path and URL addresses of web pages. To generate relevant values in service execution inputs, Snowdrop analyses word semantics using Word2Vec [79] and the Support Vector Machine [21] classifier on the fields of messages.

To test vendor-specific Android services, Iannillo et al. [53] proposed a gray-box fuzzing tool called *Chizpurfle*. By analysing the methods of services, Chizpurfle generates inputs as parameters of each targeted method. Its fuzz operators support various types of parameters, including primitive types (e.g., Boolean and float), strings, arrays/lists, and object. To increase test coverage, Chizpurfle can dynamically tune its *Fuzz Input Generator* according to the latest coverage measurement.

Table 6. Data-Driven Context Simulation Approaches Using Multiple Ways of Data Acquisition

Approach	Data Acquisition	Supported Data Type
Griebe et al. [43]	External source	Sensor
	Embedded math model	
Dynodroid [72]	Recording	GUI action, system event
	Machine generation	
MobiCoMonkey [6]	External source	GUI action, network
	Random generation	
AppDoctor [48]	External database	Text input
	Random generation	GUI input
	Heuristic search	GUI action, sensor, network, IPC, storage
ERVA [52]	VALERA	GUI action, network, image, audio, IPC Sensor, location, random number, I/O
	Event analysis	Event dependency graph (EDG)
	EventRacer [11]	Race report
RacerDroid [100]	Dynamic analysis	Statement execution
	API interception	Sensor, network, IPC, random number

Also based on fuzz testing, Qui-Gon Jinn (QGJ) [115] aims to simplify the testing of applications on wearable devices. Towards a targeted program of an application, QGJ can systematically inject a large number of mutated inter-component communication messages (intents) and GUI events by two of its components: QGJ-Master and QGJ-UI. However, QGJ cannot simulate interactions between smartphones and wearables, focusing only on context simulation inside wearables.

To test WiFi P2P applications for Android (also known as WiFi Direct [3]), Meftah et al. [74] presented a simulator called *ANDROFLEET*. Testers can define WiFi P2P scenarios by applying test scripts and installing the target application on each device. Then, ANDROFLEET can generate a large number of emulated devices that interact with each other in the intended scenarios, such as scanning peers, building/closing connection, and point-to-point communication.

3.1.4 Hybrid Approaches. Several approaches aim to cover more data types or improve the efficiency of context simulation by providing more than one method of data acquisition. Table 6 lists data-driven context simulation approaches that receive data in multiple ways.

Griebe et al. [43] presented an approach to simulate sensor input for user acceptance tests of mobile context-aware applications. By extending a UI testing tool Calabash-Android [15], this approach can generate sensor values as test cases written in the Gherkin language. Testers can edit Gherkin scripts to include explicit sensor data from external data sources. In addition, for higher abstraction in test cases, this approach contains a mathematical model that generates data by parsing human language expressions (e.g., "I shake the phone").

Machiry et al. [72] proposed an input generator Dynodroid that also supports both automated and manual generation. Testers or Dynodroid can interact with the targeted app to trigger UI and system events, including broadcasts and system service usage (e.g., location service). Testers can use console commands to switch between human and machine mode.

A similar design is also adopted by MobiCoMonkey [6]. MobiCoMonkey can inject contextual events such as network status, network delay, key events, and screen orientation changes. To provide contextual events, testers can either import a predefined context scenario file or let MobiCoMonkey automatically generate random values with a given seed for result reproduction.

Hu et al. [48] designed the cloud-based automated testing framework AppDoctor. It injects actions into an app to explore its possible executions. It can inject GUI gestures, network states,

intents, and the changes of device storage (e.g., removal of an SD card). For user input such as text boxes, AppDoctor can recognise its type (e.g., email address) and assign a corresponding value imported from an external database.

Beyond the testing of deterministic processes, Hu et al. [52] designed the tool ERVA to detect and reproduce concurrency bugs caused by races. ERVA acquires test data from three sources:

(1) *VALERA* [49, 51]: ERVA reads the contextual data of a race from the log created by the record-and-replay tool VALERA. The contextual data may comprise GUI gestures, IPC, sensor data, and I/O (e.g., file access).
(2) *Event analysis*: ERVA constructs an event dependency graph (EDG) describing the causal relationship by analysing the captured events. During replay, ERVA explores races by flipping the execution sequence of events according to the EDG.
(3) *EventRacer* [11]: Every time ERVA runs the application, EventRacer generates a report of potential races. During verification, ERVA classifies race reports to find harmful races.

Similarly, Tang et al. [100] proposed RacerDroid to detect concurrency bugs in Android applications. Given the source code of an application, RacerDroid constructs a state space model using dynamic analysis. Then, RacerDroid generates test cases to explore the potential race with two statements. Meanwhile, RacerDroid also collects contextual data comprising sensor readings, network traffic, intents, and random numbers by API interception. Finally, RacerDroid includes the captured contextual data into the test case being generated for future simulations.

3.1.5 Summary. Test data acquisition is the start of a data-driven context simulation process. Recording context data is suitable for common context scenarios, such as physical activities. When there are third-party datasets available, reusing context data for simulation is a convenient solution. Otherwise, context simulation should use manually or automatically generated data. When selecting ways of test data acquisition, testers should consider the test resources (e.g., available datasets, available data recording tools), the types of context data, and scenarios for applications.

3.2 Test Data Refinement

Test data refinement comprises three strategies: no refinement, reduction, and augmentation. Table 7 summarises these strategies for all presented data-driven context simulation approaches.

3.2.1 No Refinement. Most record-and-replay approaches replay the recorded data without any refinement. As a result, to achieve sufficient test coverage without redundancy, testers must ensure that the targeted application operates in all intended contexts during recording. In addition, most context simulation approaches based on external data do not refine the imported data. When importing external data, testers should check what context a dataset reflects.

In contrast, although some approaches that automatically generate test data do not refine the generated data, they optimise the generation process when they generate test data. For example, adaptive random test case generation [67] attempts to diversify test cases by maximising the distance between selected test cases from a pool. Aiming to detect and reproduce concurrency bugs, RacerDroid [100] only generates test cases for potential data races between two statements. Moreover, Snowdrop [122] generates relevant values of input fields using natural language processing (NLP) techniques that infer word semantics of input fields.

3.2.2 Reduction. To avoid redundant test data and to improve efficiency, several approaches employ a test data reduction technique. For example, Adamsen et al. [2] proposed a simple reduction approach to avoid redundant injections of events. When testing an application, it maintains

Table 7. Test Data Refinement of Data-Driven Context Simulation Approaches

	No Refinement	Reduction	Augmentation
Record-and-Replay	RERAN [33], MobiPlay [87], VALERA [49, 51], ODBR [81], DroidForensics [119], Paranoid [85]	MoTiF [34]	
External Data	Hermes [95], PUMA [44], Node.fz [22], MobileTest [12], TestAWARE [70], CamTest [71]	Thor [2], Caiipa [62]	Amalfitano et al. [4], KnowMe [13], Hu et al. [50]
Self-Generation	Snowdrop [122], Liu et al. [67], VanarSena [89], CATLES [30], QGJ [115], Majchrzak and Schulte [73], ANDROFLEET [74], RainDrops [123], Usman et al. [106]	Song et al. [98]	ATT [76], Chizpurfle [53]
Hybrid	Griebe et al. [43], AppDoctor [48], MobiCoMonkey [6], ERVA [52], RacerDroid [100]	Dynodroid [72]	

a history of previous abstract states and skips the injections of the events that lead to previously observed states.

In Song et al. [98], the generated test cases increase exponentially as contextual data sources increase. To maximise the test coverage in limited test runs, a two-level strategy is used to prioritise context conditions of test cases. The first phase considers test objectives, including normal or unacceptable behaviours. When testing normal behaviours, more weight is assigned to regular contexts (e.g., the SD card has space). When testing unacceptable behaviours, exceptional scenarios (e.g., the SD card is full) have more weight. In the second phase, three criteria are considered: frequency, user controllability, and minimal resource requirement. Frequency measures how many times a data source is requested in the application permission list. Sources with higher frequency are prioritised. User controllability distinguishes user-controlled data sources from system-controlled data sources. User-controlled data sources are prioritised to test applications with explicit UIs. Minimal resource requirement identifies a least number of data sources that can satisfy a permission request. For example, the request of accessing precise location may be supported by {GNSS and WiFi} or {GNSS, WiFi, and radio}. In this case, the minimal resources required are GNSS and WiFi, which gain more weights because of their necessity.

Similarly, the cloud-based framework Caiipa [62] contains the module *ContextPrioritizer* to prioritise test cases. Among all test cases, *ContextPrioritizer* aims to search for unique per-app sequences which are most likely to cause problems. Given a test app, *ContextPrioritizer* analyses previously tested apps to find a set (called *AppSimSet*) of apps with most similar behaviours. Then, *ContextPrioritizer* mines the history of test cases executed by each app in AppSimSet. Finally, *ContextPrioritizer* gives high priority to test cases that caused more problems in AppSimSet.

As a crowdsourcing-based test platform with numerous real-world users, MoTiF [34] selects test cases by aggregating crash reports from these users. The importance of a test case is also measured by its frequency of causing crashes. After collecting crowd data, MoTiF builds a crowd crash graph (CCG) which is modelled by a Markov chain representing a set of traces for an exception. Then, within a CCG, MoTiF deduces the shortest path of a crash using graph traversal algorithms. Moreover, MoTiF can identify test cases for device-specific crashes by considering both static and dynamic environments. Static contexts are constant properties of devices, such as manufacturers

and device models. Dynamic contexts are context changes including network disconnection and memory running out.

The input generator Dynodroid [72] contains a module *SELECTOR* that supports three selection strategies. The first strategy, *Frequency*, gives higher priority to least generated events and least used UI widgets to increase state coverage. The second strategy, *UniformRandom*, randomly selects candidate test cases. However, these two strategies cannot effectively find critical test cases relevant to certain contexts. To address this issue, the third strategy, *BiasedRandom*, counts frequency in a context-sensitive way. Every time an event is selected in a certain context, *SELECTOR* reduces the chance of selecting it again in such a context.

3.2.3 Augmentation. Considering that the original test data may be insufficient, several approaches are able to generate more test data for higher test coverage.

During contextual data replay, KnowMe [13] provides an option to mix the original data with Gaussian noise generated by an internal function. By adjusting the distribution parameters of Gaussian noise, testers can simulate different noise levels in contexts to examine how noise affects the behaviours and performance of the tested application.

Similarly, Hu and Neamtiu [50] proposed SSDA which can automatically create semantically meaningful data by modifying captured sensor data. SSDA supports the alteration of geo-locations, audio, and images. For geo-locations, SSDA has three alterations: null, map shift, and speed change. Null location represents the unavailability of location access due to poor signal or the closure/malfunction of hardware modules. Map shift and speed change are straightforward. They modify values of location coordinates and the speed of coordinate change within a period. To test microphone-based apps, SSDA can alter audio streams by changing the sampling rate and adding background noise. For apps analysing photos, SSDA transforms an original image by adjusting brightness, colour balance, size, blur, and rotation.

In Amalfitano et al. [4], a mutation-based technique is used to extend existing test cases. It generates new test cases by embedding additional sequences of contextual event patterns into existing test cases. For example, to test the robustness of a smartphone app, testers can add an event of an incoming phone call when the tested app is operating in a specific context defined by an existing test case. Testers can apply this technique either manually or using automated tools.

Another mutation-based technique is used in Chizpurfle [53]. This technique aims to generate new test cases that are likely to drive the app to execute more new blocks in source code. Given a seed (i.e., the initial test case), the test case generator of Chizpurfle constructs a new test case by mutating the seed using fuzzing tools for different data types. Chizpurfle measures the number of new code blocks executed after test cases are input. If a test case leads to the execution of new blocks, Chizpurfle will use it as a seed to further generate new test cases.

In addition to static components, ATT [76] can generate data for dynamic data requests. As applications may register system services or build IPC to request momentary contextual data at runtime, ATT can detect these dynamic requests to generate corresponding data on the fly.

3.2.4 Summary. Test data refinement can be a critical step, as the amount and quality of test data can improve the efficiency and effectiveness of tests. Due to the context heterogeneity of various mobile context-aware applications, refinement approaches have many unique data processing mechanisms for different context types. Some data-driven context simulation approaches already embed refinement into their data generation process. For other approaches, sufficient reduction or augmentation of original data may significantly improve test performance and results.

3.3 Execution of Context Simulation

Table 8 summarises the execution characteristics of these approaches.

Table 8. Execution Characteristics of Data-Driven Context Simulation Approaches

Approach	Test Data Delivery	Simulator Speed	Operating Environment	OS Modification	App Modification
MobiPlay [87]	Event injection	Original, fast	Device+server	Android OS	No
CrashScope [82]	Debug command	Original	Device+cloud	No	No
DroidForensics [119]	Debug command	Uncontrolled	Device+cloud	Android SDK	No
Hu et al. [50]	Event injection	Original	Device	Android OS	Bytecode
VALERA [49, 51]	Event injection Task scheduling	Original	Device	Android OS	Bytecode
Chizpurfle [53]	Event injection	Uncontrolled	Device	No	Bytecode
QGJ [115]	Event injection Debug command	Uncontrolled	Device	No	No
TestAWARE [70]	Event injection	Slow, original, fast	Device	No	Source code
CamTest [71]	Event injection	Slow, original, fast	Device	No	Source code
KnowMe [13]	Event injection	Slow, original, fast	PC	No	Source code
Amalfitano et al. [4]	Event injection	Uncontrolled	Device	No	No
Majchrzak and Schulte [73]	Code injection	Uncontrolled	Device	No	Source code
Node.fz [22]	Task scheduling	Original	PC	Node.js (libuv)	No
Song et al. [98]	Debug command	Uncontrolled	Device	No	No
Usman et al. [106]	Debug command	Uncontrolled	Device	No	No
ANDROFLEET [74]	Debug command	Uncontrolled	PC	Android SDK	No
MobileTest [12]	Debug command	Uncontrolled	PC	No	No
Liu et al. [67]	Debug command	Uncontrolled	PC	No	No
Caiipa [62]	Event injection	Uncontrolled	Cloud	Windows OS	No
Griebe et al. [43]	Event injection	Original	Device	Android SDK	No
Hermes [95]	Event injection	Uncontrolled	Device+server	No	No
RERAN [33]	Event injection	Original, fast	Device	No	No
MoTiF [34]	Debug command	Original	Device+cloud	No	No
RacerDroid [100]	Event injection Task scheduling	Uncontrolled	Device	Android SDK	Source code
ATT [76]	Debug command	Original	PC+cloud	No	Bytecode
RainDrops [123]	Event injection	Original	Cloud	Android OS	No
CATLES [30]	Event injection	Original	Device+cloud	No	Source code
Snowdrop [122]	Event injection	Uncontrolled	PC+device	No	No
Paranoid [85]	Task scheduling	Original	Device+cloud	Android OS	No
ERVA [52]	Task scheduling	Uncontrolled	Device	Android OS	No
VanarSena [89]	Event injection	Uncontrolled	Cloud	No	Bytecode
Dynodroid [72]	Debug command	Uncontrolled	Device	Android SDK	No
PUMA [44]	Debug command	Uncontrolled	Device	No	Bytecode
AppDoctor [48]	Event injection	Slow, fast	Cloud	No	Bytecode
Thor [2]	Code injection	Uncontrolled	Cloud	Android OS	Source code
MobiCoMonkey [6]	Debug command	Original	Emulator	No	No

3.3.1 Test Data Delivery. An essential feature of a data-driven context simulation approach is its way to deliver test data to targeted applications. For different test purposes, context simulation approaches employ various ways of test data delivery. Several approaches use more than one way of data delivery to support multiple test purposes.

Debug command. Mainstream mobile platforms provide versatile command line tools for testers to perform tests on emulators or actual devices. For example, Android has its debug infrastructure ADB [35] which can control a suite of preinstalled testing tools. To simulate generic scenarios, many context simulation approaches deliver test data by invoking debug commands. An advantage of using debug commands is the simplicity in implementing context simulators. With light-weight design and implementation, MobiCoMonkey [6] simulates network conditions and several GUI actions on Android emulators using ADB commands. CrashScope [82] also uses ADB to deliver GUI actions and sensory values (only for emulators) via *adb input* commands and telnet connections.

In addition to official command line tools from mobile platforms, several third-party testing tools have customised command lines that can be invoked or extended by context simulators. For instance, MoTiF [34] and ATT [76] feed events into apps using the Android testing tool Robotium [90]. ANDROFLEET [74] introduces a novel vocabulary to define WiFi Direct scenarios based on Calabash-Android [15]. During simulation, the host machine loads Calabash-Android to parse scenario definition files that configure a cluster of emulators. Based on SQL commands, DroidForensics [119] provides an interface that receives SQL-like queries for post-mortem examination of cyber attacks. Such examination operates on a cloud server with data mining algorithms to generate a causal graph that summarises the contextual information.

Event injection. As most mobile context-aware applications are event-based applications, event injection is a popular way adopted by context simulation approaches. For example, Snowdrop [122] and Chizpurfle [53] automate the testing of background services by mocking apps or the OS to send inter-component messages which contain contextual information.

To simulate sensor data on Android without modifying the targeted app, several tools, such as RERAN [33] and MobiPlay [87], inject values into communications between hardware drivers and the Linux kernel of Android OS. However, such injection requires root privileges and cannot work for all types of contextual data. In RainDrops [123] and Griebe et al. [43], event injection is achieved by modifying API classes in Android SDK (Software Development Kit) so that all types of events can be simulated to unmodified apps. For Windows 8 and Windows Phone 8 apps, Caiipa [62] modified the GNSS driver of the Windows OS to deliver spoofed coordinates and response. In contrast, VanarSena [89] rewrites sensor API calls in the bytecode of Windows Phone apps to inject sensor values without modifying the Windows OS. AppDoctor [48] also uses this method for event injection on Android. To simulate single-camera video for camera-based Android applications, CamTest [71] requires developers to call its customised APIs, instead of standard camera APIs, for the decoding and replay of video frames from MP4 video files on a physical device.

Considering that mobile context-aware applications may collect contextual data from third-party data collection tools, several context simulation approaches deliver data by simulating third-party tools. For instance, KnowMe [13] and TestAWARE [70] deliver data in the format of inter-component communication messages defined by AWARE [25]. Hence, AWARE-based applications can receive data from them without any modification. To test other applications with them, testers have to modify the source code of data collection to receive AWARE format messages.

Code injection. Code injection is widely used in cyber attacks. Rather than launching attacks, Thor [2] and Majchrzak and Schulte [73] inject code into the source code of apps to change contextual information at runtime. The purpose is to validate how different contexts affect the execution

traces of apps at the source code level. However, this technique cannot be applied to closed source apps.

Task scheduling. As some mobile context-aware applications involve non-deterministic processes (e.g., multithread and asynchronous tasks), events and executions are often randomly interleaved. The preceding context simulation approaches may fail to reproduce a concurrency bug (e.g., race conditions) caused by certain context. Hence, several approaches provide deterministic simulation by modifying task scheduling processes of OS or environments so that testers can explore all possible schedules of events to ensure the reproduction of a concurrency bug.

On Node.js servers such as for IoT devices, Node.fz [22] simulates events in a deterministic way by organising the sequence of three kinds of non-deterministic events: external input, callback responses, and worker pool scheduling. To achieve this, Node.fz replaces the corresponding modules in the libuv [63] library which arranges asynchronous events and I/O controls in Node.js.

On Android, Paranoid [85] schedules threads and I/O control system calls using a light-weight scheduler in the user space of the Linux kernel. By sending system calls that manages locking of threads, Paranoid can prevent threads with shared memory from running concurrently. Regarding non-deterministic network traffic, VALERA [49, 51] modifies APIs of HTTP/HTTPS connection in the Android SDK to feed apps data from its log rather than real networks. To detect data races between two statements, RacerDroid [100] directly controls the dispatch sequence of the statements by extending the event dispatching functions of the automated testing tool Espresso [39].

3.3.2 Simulator Speed. Numerous mobile applications analyse time-series data to recognise context. Thus, delivering contextual data to apps at right time is the key to high fidelity in context simulation. However, testers may want to run quick simulations with longitudinal datasets which may last for several weeks. They may also want to slow down simulations to thoroughly examine complex programs which may take some seconds to complete each action. To achieve these goals, researchers have designed several methods to control the speed in context simulation.

Original speed. It is technically challenging to deliver events at timings with microsecond accuracy [33]. As the *sendevent* tool in the Android debug command has a lag when releasing continuous events, RERAN [33] and MobiPlay [87] choose to inject events through hardware driver interfaces which can trigger signals instantly. VALERA [49, 51] delivers events with modified Android APIs. It is found that replaying recorded data from log files is faster than normal data collection with standard APIs. Hence, VALERA lets the replay process sleep for the same amount of time as standard API calls. During video replay, CamTest [71] can follow the original speed of a video file by referring to the video frames per second. Delivering contextual data via inter-component messages, KnowMe [13], TestAWARE [70], and Paranoid [85] modify the original timestamps by considering current device clocks and time differences in consecutive data samples.

Acceleration and slow motion. To accelerate simulations, RERAN [33] and MobiPlay [87] enable fast forwarding by compressing time in idle periods between two events. On KnowMe [13] and TestAWARE [70], testers can set an arbitrary speed multiple v for slow, normal, or fast simulation. Given the original time difference t in two consecutive data samples, the two simulators wait for time t/v to achieve the speed setting. Moreover, for audio streams, TestAWARE takes into account the sampling rate of audio files (denoted as s) by using $\frac{1}{s \times v}$ as the wait time between two consecutive audio frames. CamTest [71] also allows the targeted application to change the speed of video replay by interrupting or accelerating the replay process using its customised API. In contrast, AppDoctor [48] handles acceleration and slow motion differently for each event. It has two modes: approximate and faithful. The approximate mode accelerates simulations by invoking high-level APIs that contain multiple executions, whereas the faithful mode injects each event and

waits for an event-specific period of time before next event. Hence, the faithful mode can correlate events and responses from time-consuming executions.

Uncontrolled. Many context simulation approaches do not explicitly control the simulation speed. One reason is that some approaches do not simulate time-series data. For instance, focusing on testing background services, Chizpurfle [53] invokes each targeted method and analyses the output in a round-by-round process. Similarly, using code injection, Majchrzak and Schulte [73] and Thor [2] execute one statement and one assertion at a time during simulation. In addition, although without speed control, some approaches consider the timing of events. For example, VanarSena [89] simulates impatient users by injecting the next event before the response for the last event.

3.3.3 Operating Environment. To a large extent, the operating environment determines the effectiveness and efficiency of context simulation. Numerous approaches perform context simulation on physical devices. Many of them are record-and-replay approaches (e.g., RERAN [33] and VALERA [49, 51]) because the data must be captured on physical devices first. In addition, performing simulation on physical devices can help testing tools find performance issues, as PC-based emulators are often high performance [70]. In addition, certain apps, such as games, can only run on mobile devices with ARM processors, because x86 processors on PCs cannot execute code from ARM-oriented libraries. For apps supporting multiple architectures such as x86 and ARM, several approaches conduct context simulation on PC or PC-based emulators. This method is convenient for testers who do not own a suitable physical device. Exemplar approaches include MobiCoMonkey [6] and Node.fz [22], as Android and Node.js are designed for different kinds of hardware devices. Android emulators especially support injections of many data types, such as GNSS and network states, from debug commands. However, Android debug commands do not offer this feature on physical devices. Another advantage of context simulation on PCs is that testers can simultaneously run multiple instances of emulators. On a PC, ANDROFLEET [74] simulates a cluster of Android emulators to test WiFi P2P apps.

The cloud is also a popular operating environment of context simulation approaches. Testing resources are cost efficient and easily expandable with cloud infrastructures. For example, based on a private cloud with virtual processor cores or physical devices, ATT [76] can launch context simulation tasks on emulators or physical devices in parallel to increase test coverage. RainDrops [123] also relies on a cloud of physical devices and emulators where emulators offload the sensor data collection to physical devices for high fidelity. Another advantage of cloud is that testing tools can learn from previously tested apps and test cases recorded in cloud. Given an app, Caiipa [62] prioritises its test cases by mining the history of similar apps and test cases.

3.3.4 OS Modification. Many context simulation approaches require testers to customise OS. The benefit of OS modification is that testers can simulate all data types for unmodified apps. However, the installation of a modified OS may be a time-consuming task for common testers.

Although cannot directly operate on commercial devices, some approaches only need root privileges which can be obtained from an unmodified OS. For example, root privileges allow RERAN [33] and MobiPlay [87] to inject events into hardware driver interfaces. Root privileges can also allow the installation of privileged programs as light-weight OS modification. Paranoid [85] is implemented as a privileged user application in the user space of Linux kernel below Android. However, many approaches rely on customised OS components.

More complexly, some approaches (e.g., RainDrops [123] and Griebe et al. [43]) overwrite sensor APIs in OS SDK. Moreover, several approaches instrument hardware drivers (e.g., GNSS driver modification by Caiipa [62]) or task schedulers (e.g., event and worker scheduling modification by Node.fz [22]).

3.3.5 App Modification. Some context simulation approaches have to operate with modified apps. Such modification can be at two levels: bytecode and source code. Bytecode modification is a popular method, as many apps are distributed as a package of bytecode. For example, VALERA [49, 51] rewrites bytecode of Android apps for sensor data interception. Chizpurfle [53] modifies bytecode of Android services to track the test coverage on each code block. ATT [76] and AppDoctor [48] instrument the package of Android apps by gaining network permission for controller-app communication and unifying signatures for sharing memory with tested apps.

Comparatively, few approaches involve modification at the source code level. To deliver audio streams, TestAWARE [70] wraps audio frames inside Android Intent messages that can only be received by apps with an extra corresponding message listener. CamTest [71] has a package of its customised APIs that can replace standard camera APIs to deliver video frames to targeted applications. RacerDroid [100] instruments the statements in source code for purposely deterministic executions to detect data races. Source code modification is also adopted by Thor [2] and Majchrzak and Schulte [73] for tests with injected statements and assertions.

3.3.6 Summary. As mobile context-aware applications are designed for various running environments, the execution requirements of data-driven context simulation approaches significantly vary. Many approaches require testers to modify OS or applications for test data delivery. Some approaches rely on specialised servers or the cloud. Hence, certain system configuration knowledge and programming skills are needed for the execution of data-driven context simulation.

4 MODEL-BASED CONTEXT SIMULATION

We now turn our attention to model-based context simulation. As in the previous section, we consider the three phases: application modelling and acquisition of test cases (Section 4.1), refinement (Section 4.2), and execution (Section 4.3).

4.1 Application Modelling and Acquisition of Test Cases

For test case acquisition, model-based context simulation approaches typically have to first build an abstract model to describe the tested application with its operating context. Table 9 summarises the modelling and test case acquisition mechanisms of model-based context simulation approaches.

Yu et al. [118] presented an approach that involves two different models. It uses an Extended Finite State Machine (EFSM) [17] to model sensors and a Biographical Reaction System (BRS) [80] to model the environment. Beyond a regular finite state machine, an EFSM uses variables, operations, and transition conditions to describe a sensor. To model multiple sensors, they build an overall EFSM by the Cartesian product of each single EFSM. To model the environment, a BRS consists of both static and dynamic properties about a bigraph. A bigraph is represented by a place graph with locality or containment about entities and a link hyper-graph with relationships between entities. According to reaction rules, a Bigraphical Labeled Transition System (B-LTS) can be created for a BRS. Using the Cartesian product of an EFSM and a B-LTS, test cases can be structured. To avoid the exhaustive generation of all possible test cases, they proposed a pattern-flow-based strategy to generate test cases only for specific patterns in reactions.

Regarding different phases in the testing process, Wang et al. [109] proposed the Model-based Simulation Testing Framework (MSTF) comprising five models: the SUT model (abstraction of software under test), environment model (abstraction of physical surrounding around software), test case model (specifications of input and output data), test execution model (specifications of hardware and software resources), and test conclusion model (specifications of potential correlation between faults and expected test results). MSTF allows testers to determine the format of these

Table 9. Modelling and Test Case Acquisition of Model-Based Context Simulation

Approach	Application and Context Modelling	Test Case Acquisition
Yu et al. [118]	BRS (app model), EFSM (sensor model)	Pattern-flow generation strategies
MSTF [109]	SUT model, environment model, test case model, test execution model, test conclusion model	External data sources
Liñán et al. [64]	Augmented model	Automatic ripping
Zhang et al. [124]	Graph-based test model	External data sources
SIT [86]	Interactive app model	Sampling-based generation strategy
Wang et al. [110]	Function call graph, control flow graph	Enhancing existing test suites
Griebe and Gruhn [42]	Unified Modeling Language Activity Diagram	Petri net trace analysis
Micskei et al. [78]	Context model, scenario model	Search-based generation strategy
Sama et al. [94]	Adaptation Finite-State Machine	Not required
Xu et al. [114]	Adaptation model	Manual definition
Tönjes et al. [102]	EFSM	Transition coverage, transition and state-coverage search
Ben Abdessalem et al. [1]	ADAS formalisation	Multi-objective search, decision tree classification
Garzon and Hritsevskyy [31]	Discrete event system specification (DEVS)	Stochastic DEVS
JPF-ANDROID [107]	JPF-generated Promela model	Exhaustive generation
DroidPF [8]	JPF-generated Promela model	Incremental driver generation

models. A weakness of MSTF is that it does not support automated test case generation. Testers must import test cases and corresponding expected results from external data sources.

Similarly, Liñán et al. [64] proposed another multi-model concept called the *augmented model*. An augmented model comprises context (e.g., sensors and hardware states), domain (e.g., application data, entities and relations), usage (e.g., user interactions), and GUI (e.g., graphic components) information. The augmented model can be generated automatically by ripping and static code analysis. Using the augmented model, the context states can be simulated by ripping triggers.

Targeting location-based mobile apps, Zhang et al. [124] introduced a novel graph-based test model that can be denoted by $G = (N, E, S, Lc)$, where G is a graph model, Lc is the location context map, N are map nodes (e.g., buildings, app users, and traffic jams), E is service execution (e.g., one node detects another), and S is the service set for all nodes. Although this model can reflect dynamic contexts and app states, it lacks automation support. Testers need to use external data sources to obtain test cases.

Qin et al. [86] presented an approach called *Sampling-based Interactive Testing* (SIT) which uses an interactive app model (IAM) to specify the characteristics of the app, environment, and interactions in between. IAM also considers uncertainties of interactions, such as unreliable sensor values and flawed physical actions (as defined in Ramirez et al. [88]). To generate test cases, SIT adopts a sampling-based strategy which tracks execution traces of input and derives new input from previous input with low similarity in execution traces. SIT is implemented in Java 8 and can automate test case generation. Although the source code of the target application can be directly included into an IAM, testers must manually specify the rest part of the IAM according to the documentation of the app.

Aiming to enhance existing test suites, Wang et al. [110] developed an approach that automatically generates variant test suites by identifying statements associated to context changes. From

the source code of an app, it generates the function call and control flow graphs (CFG) to find such statements (marked as nodes in CFGs). Based on the flows containing these CFG nodes, it searches for variant test suites involving context conditions that cause different app behaviours.

Griebe and Gruhn [42] proposed an approach that can automatically parse models in Unified Modeling Language (UML) [92] to generate test cases. Beyond regular UML Activity Diagrams (ACD) context, they associate ACD elements and context parameters using UML stereotype classes in a UML profile. Then, each ACD is converted into a Petri net to simplify test coverage quantification and to generate test cases represented by traces.

Micskei et al. [78] described a method which allows testers to specify an app by a context model in OWL2 ontology and a scenario model in UML Sequence Diagrams. A context model defines static properties and dynamic relations about environment objects. A scenario model describes an app's input events and output actions. Based on the two models, a search-based technique is used to automate test case generation. The search strategies can find minimal context objects required for a scenario requirement, combined context objects related to multiple requirements, and context objects with boundary values.

To model applications adapting to context, Sama et al. [94] defined an Adaptation Finite-State Machine (A-FSM) denoted as $M = (\mathcal{S}, \delta, S_{initial}, S_{final})$, where $\mathcal{S}$ denotes all possible states of an app, $S_{initial}$ the initial state, S_{final} the final state, and $\delta \subseteq \mathcal{S} \times \mathcal{R} \times \mathcal{S}$ the transition relation. δ can be derived from adaptation rules in set $\mathcal{R}$ by $\delta = \{(S, R, S_0) | \exists R = (S, P, S_0, A, N) \in \mathcal{R}\}$. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S} \times \mathcal{A} \times \mathbb{N}$, where $\mathcal{P}$ denotes logical predicates on propositional context variables, $\mathcal{A}$ the actions of adaptations, and $\mathbb{N}$ the natural numbers representing rule priorities. Based on A-FSM, they use five algorithms to automatically detect two kinds of faults: behavioural faults caused by wrong rule predicates and context hazards caused by asynchronous updates of multiple context variables. These algorithms operate by exploring rules and states in A-FSM so that test cases are not required.

Moreover, Xu et al. [114] designed an Adaptation Model (AM) to express complex adaptations of apps. An AM is defined as $AM = (S, R, s_0, S_f, V, s_c)$, where S represents the state set, R the adaptation rule set, s_0 the initial state, S_f the final state set (apps stop at any state in S_f), V the assignment of all contextual variables, and s_c the current state. Compared to approaches based on propositional logic, an AM can describe complex rules in adaptation using a customised language based on first-order logic. In addition, an AM can detect faults expressed by dynamic transitions between states. To test adaptations of an app, an algorithm takes as input the AM and context changes specified by testers to detect faults.

To automate modelling of an app, Tönjes et al. [102] proposed a tool that reads application specifications in Web Application Description Language (WADL). Based on WADL files, this tool constructs an Application Behaviour Model (ABM) represented by an EFSM which contains app states, context events, app actions, and transitions between states. By searching for traces from the initial state to the final state in an EFSM, this tool generates test cases in the TTCN-3 language that can be executed on simulators.

For testing Advanced Driver Assistance Systems (ADAS), Ben Abdessalem et al. [1] provided an evolutionary search approach that can explore complex input space using a formalised ADAS model. The ADAS formalisation considers both the autonomous vehicular system and environment, including states of pedestrians and vehicles. The test case generation first produces a set of context scenarios using a genetic algorithm. Then, these context scenarios are used to test the system behaviours which may fail to ensure safety. A decision tree learns from the results to identify critical regions in the ADAS input space.

For personalised context-aware applications, Garzon and Hritsevskyy [31] proposed a model-based context simulator which produces recurrent user behaviour using discrete event system

specification (DEVS) [120]. Each model represents a scenario-specific event pattern of a human user. Using stochastic DEVS [57], for both deterministic and non-deterministic scenarios, this approach is able to generate recurrent user behaviour as sensor readings with timestamps.

For Android apps, Van der Merwe et al. [107] designed a verification tool called *JPF-ANDROID* by extending Java PathFinder (JPF) [45]. JPF can translate Java code into a model in Promela language for the SPIN model checker [47] which performs simulation to examine properties such as absence of deadlocks and races. JPF-ANDROID automatically extracts key components (e.g., activity, service, and message queue) of Android apps to rewrite the apps in standard Java code. In addition, JPF-ANDROID allows users to specify system events written in scripts. During the verification process, JPF-ANDROID generates all possible combinations of system events in exhaustive search to explore the states of the abstract app model.

In addition, Bai et al. [8] proposed another JPF-based verification tool, DroidPF, to verify security properties (e.g., correct transmissions of sensitive data) of Android apps by a driver program to explore the state space. According to a property specified by testers, DroidPF extracts related components and events by statically analysing source code. Then, DroidPF explores the states of the extracted modules with an mock-up Android OS on JPF. To take into account user interaction and contextual input, DroidPF can construct drivers that simulate and schedule such events to perform corresponding state transitions during exploration. Moreover, with an incremental mechanism, DroidPF can also generate ad hoc drivers to explore components that are newly created on the fly.

4.1.1 Summary. Most mode-based context simulation approaches rely on specialised modelling methods, unlike a few approaches using common modelling methods such as UML and Promela. Testers often need to manually create application and context models using specific software modelling skills. In addition, most approaches can automatically generate test cases, although several approaches require testers to provide test cases.

4.2 Refinement of Test Cases and State Space

Model-based context simulation approaches may refine their test cases or state space before test executions. Some of them remove low-quality test cases or simplify the state space of a model to improve test efficiency. Some generate more relevant test cases to augment original input to increase test coverage. Table 10 lists these refinement methods of model-based context simulation.

Several approaches do not refine test cases or state space, including MSTF [109], Zhang et al. [124], Xu et al. [114], Micskei et al. [78], and JPF-ANDROID [107]. Among these, refinement is not necessary in Micskei et al. [78] since the test case generation phase already minimises the number of test cases to sufficiently cover a functionality.

The remaining approaches all make an effort to refine test cases or state space. In Yu et al. [118], testers have two options to select relevant test cases: all-defs and all-uses. All-defs aim to test every pattern with some test cases, whereas all-uses find a sufficient set of test cases to explore every pattern use.

SIT [86] optimises both test cases and state space by pruning. Inside the input space, it removes redundant test cases that are shared by multiple partitions. In addition, it discards redundant traces that involve an identical set of input at highly similar states of the target model.

In Wang et al. [110], a context manipulator augments original tests cases by generating new context data that leads to unexplored context scenarios.

In Griebe and Gruhn [42], test case generation relies on the automatic analysis of the reachability graph of a Petri net which represents an application model with stereotypes for context description in a UML profile. First, all paths from the tree root to leaves are selected as potential test cases.

Table 10. Test Case Refinement and State Space Refinement of Model-Based Context Simulation Approaches

Approach	Test Case Refinement	State Space Refinement
Yu et al. [118]	Pattern-flow testing	No
MSTF [109]	No	No
Liñán et al. [64]	No	No
Zhang et al. [124]	No	No
SIT [86]	Pruning	Pruning
Wang et al. [110]	Context manipulation	No
Griebe and Gruhn [42]	UML stereotype-based reduction	No
Micskei et al. [78]	No	No
Sama et al. [94]	No	State matrix-based simplification
Xu et al. [114]	No	No
Tönjes et al. [102]	Similarity analysis	No
Ben Abdessalem et al. [1]	Decision tree learning	No
Garzon and Hritsevskyy [31]	No	No
JPF-ANDROID [107]	No	No
DroidPF [8]	Dependency-constrained event permutation	Slicing-based reduction

Then, in each path, the refinement process removes the nodes that are not associated to UML stereotypes. Considering that the resulting paths may have duplicates, the refinement process discards all but one path within these duplicates.

In Sama et al. [94], a derivative representation, named *state matrix*, is proposed to simplify the state space of models. A state matrix contains sets of truth assignments of context variables and sets of associated rules. A state matrix excludes state assignments that are not related to any associated rule.

In Tönjes et al. [102], similarity analysis is used to reduce the number of highly similar test cases. It computes a pairwise similarity score between two test cases using the Levenshtein distance. Given a target number, it adopts a greedy search to find a set of test cases with the lowest average similarity between pairs of test cases.

In Ben Abdessalem et al. [1], a decision tree is trained with initial context scenarios and test results to identify critical regions in the input space. These critical regions contain context scenarios that are more likely to expose failures of ADAS.

DroidPF [8] refines both test cases and state space. Given the source code of an Android application, DroidPF applies a slicing-based reduction to extract an executable slice using program slicing [112]. Since the sliced program contains only key components that are essential for executions or are related to security properties, DroidPF can efficiently explore the state space without loss of security vulnerabilities in verification. In addition, considering that not all event sequences are relevant to security properties, DroidPF adopts a dependency-constrained event permutation to construct only feasible event sequences as selected test cases.

4.2.1 Summary. Most model-based context simulation approaches refine test cases or state space. Since the state space of models is often large, many approaches aim to avoid time-consuming and memory-hungry exhaustive search (computers may not have enough physical memory for this) by simplifying their models or removing test cases that are likely to be redundant.

Table 11. Execution Characteristics of Model-Based Context Simulation Approaches

Approach	Deployment	Software Dependency	Operating Environment
Yu et al. [118]	Offline	Eclipse [28]	PC
MSTF [109]	Offline	App specific	Distributed system
Liñán et al. [64]	Online	ADB [35]	Device/emulator + server
Zhang et al. [124]	Offline	App specific	App specific
SIT [86]	Online	Java 8	App specific
Wang et al. [110]	Online	Java 1.4	App specific
Griebe and Gruhn [42]	Online	Calabash [15]	Device/emulator + server
Micskei et al. [78]	offline	App specific	App specific
Sama et al. [94]	Online	TestingEmulator [93]	Emulator
Xu et al. [114]	online	Cabot middleware [113]	PC+device
Tönjes et al. [102]	Online	TTworkbench [101]	Server
Ben Abdessalem et al. [1]	offline	App specific	App specific
Garzon and Hritsevskyy [31]	Offline	Java	PC
JPF-ANDROID [107]	Offline	JPF [45]	Java virtual machine
DroidPF [8]	offline	JPF [45] SAAF [46], apktool [104] dex2jar [23], Dare [84]	Java virtual machine

4.3 Execution of Context Simulation

Table 11 shows the execution characteristics of model-based context simulation approaches in terms of deployment, software dependencies, and operating environments.

4.3.1 Deployment. Model-based context simulation can be deployed in an online or offline manner for execution. Online simulation approaches execute tests on target applications, whereas offline simulation approaches operate on software/context models that represent target applications and context factors of ambient environments.

Online approaches directly interact with target applications during context simulation. An advantage of online context simulation is that future test cases can be improved on the fly according to the current output and context state. For example, sampling-based interactive testing (SIT) [86] considers the target application a component of the model to conduct sampling-based testing which optimises future input by analysing the recent result from the application at runtime. Such online context simulation is also adopted in other works [42, 94, 102, 110, 114].

As direct transmission of test cases to implemented applications may be technically impractical, some offline approaches perform simulation to generate application-executable test cases that can be downloaded for execution on implemented applications. In MSTF [109], a development node, usually a PC, manages context simulation and generates test cases. Then, the test cases are downloaded to distributed real-time executing nodes based on target hardware devices which can validate real-time requirements of applications. Similar mechanism is also used in Micskei et al. [78].

In addition, considering that executing test cases on implemented applications may not be efficient or practical, several offline approaches only perform simulation on abstract software/context models, without directly examining implemented software applications. Hence, their test cases are abstract and can only be processed by models rather than actual applications. For example, in Yu et al. [118], context simulation operates on only abstract software/context models. As well, in Zhang et al. [124], JPF-ANDROID [107], and DroidPF [8], context simulation only aims to examine the behaviours represented by models.

4.3.2 Software Dependency. To execute context simulation, approaches may have to employ other software packages. In Yu et al. [118], modelling and simulation are managed by two customised plug-ins of Eclipse [28]. Specific versions of Java are used in SIT [86] and Wang et al. [110]. Conversely, several approaches rely on third-party simulation and testing tools. Supporting the script-based simulation of Android and iOS applications, Calabash [15] is used in Griebe and Gruhn [42]. Aiming at J2ME applications, TestingEmulator [93] is adopted in Sama et al. [94]. In Xu et al. [114], Cabot middleware [113] is used for context management. Based on the cross-platform TTCN-3 language [41], TTworkbench [101] serves as the execution automation tool in Tönjes et al. [102]. To perform model checking for Android applications, JPF-ANDROID [107] and DroidPF [8] rely on JPF [45] to convert Java bytecode into Promela models. DroidPF [8] also uses SAAF [46] for static slicing, and apktool [104], dex2jar [23], and Dare [84] for decompilation. Comparatively, software dependencies are not restricted in MSTF [109], Zhang et al. [124], Ben Abdessalem et al. [1], and Micskei et al. [78], only providing abstract frameworks for testers to specialise.

4.3.3 Operating Environment. Most model-based context simulation approaches are designed for specific operating environments. In Yu et al. [118], context simulation generates offline test cases on a PC for applications to download on intended devices. Aiming at the examination of distributed embedded systems, MSTF [109] operates within Ethernet to generate test cases on a development node and execute tests on real-time execution nodes within a real-time network. In Griebe and Gruhn [42], context simulation operates on a server to interact with targeted applications on physical devices or emulators on the fly. In Sama et al. [94], context simulation is managed by a customised J2ME emulator—TestingEmulator [93]. In Xu et al. [114], context simulation operates on a PC which communicates with applications on mobile devices using a wireless network. In Tönjes et al. [102], context simulation can be conducted on a TTworkbench [101] server using a web service interface or manually. Based on JPF [45], JPF-ANDROID [107] and DroidPF [8] launch context simulation using Java virtual machines. By contrast, operating environments are not bounded in Zhang et al. [124], SIT [86], Wang et al. [110], Ben Abdessalem et al. [1], and Micskei et al. [78], because they rely on cross-platform dependencies or high abstraction levels.

4.3.4 Summary. Most model-based context simulation approaches only examine abstract models to check the correctness of software design. These types of examination can be deployed in generic computing environments, such as PCs. Meanwhile, some model-based context simulation approaches can generate concrete test cases to directly check the functionality of implemented applications. However, testers must have sufficient knowledge to configure operating environments.

5 DISCUSSION

5.1 Comparison Between Data-Driven and Model-Based Context Simulation

Table 12 shows a comparison of characteristics of data-driven and model-based context simulation. Generally, model-based approaches focus on abstract software design and are more flexible, whereas data-driven approaches aim at implementation details.

5.1.1 Popularity. According to the number of studied works, data-driven approaches are the main component of context simulation for testing mobile context-aware applications. Researchers and developers have designed various techniques for test case acquisition, test case refinement, and test execution of data-driven context simulation. These techniques cover a relatively large part of the need of testing, in terms of diversity of supported context events and examined properties of applications. Compared to data-driven approaches, model-based context simulation receives less attention from developers and researchers. But we still observed significant diversification

Table 12. Comparison Between Data-Driven and Model-Based Context Simulation

	Data Driven	Model Based
Popularity	High	Low
Test Object	Implemented software	Software model, implemented software*
Software/Hardware Dependency	Strict	Flexible
Context Type Diversity	High	Low
Non-Deterministic Test	Weak	Strong
Laborious Process	Data acquisition,* environment configuration	Model definition,* environment configuration*

*If necessary.

in application modelling, test case acquisition, test case refinement, state space reduction, and execution of model-based techniques.

5.1.2 Test Object. Regarding test objects, data-driven approaches focus only on implemented software. This implies that testers cannot perform any simulation to test a mobile context-aware application before the completion of coding activities. In contrast, most model-based approaches aim to test software models instead of actual applications. These model-based approaches operate with a higher level of abstraction of applications, since software models are created in the software design phase. Some model-based approaches also generate concrete test cases for execution on actual applications to ensure the correctness of both software models and implementation.

5.1.3 Software/Hardware Dependency. Data-driven approaches often require strict software or hardware dependencies (e.g., a certain version of a third-party library or a specific cloud server), due to the compatibility of applications and simulators. Some data-driven approaches rely on modified OS. Comparatively, model-based approaches can generally operate in different environments, although a few model-based approaches require specific environments to execute concrete test cases for the validation of actual applications.

5.1.4 Context Type Diversity. Compared to model-based approaches, data-driven approaches support a wider range of context types. Many data-driven approaches can simulate multiple types of context events, whereas a few focus only on one kind of context. Although model-based approaches consider context as states in models, they have limited ability to simulate complex context such as video and audio for the validation of low-level processing.

5.1.5 Non-Deterministic Test. Due to the technical challenges of modifying task scheduling processes of OS, only a small number of data-driven approaches can simulate context to test non-deterministic processes (e.g., random decisions and asynchronous tasks). It is worth noting that developers have no access to the source code of closed source mobile OS, such as iOS, for testing purposes. In contrast, many model-based approaches take into account uncertainty. Hence, these model-based approaches can efficiently examine non-deterministic processes.

5.1.6 Laborious Process. Although several data-driven approaches can automatically generate relevant context data for tests, most data-driven approaches require testers to record or import test cases. When there is no previous or third-party data recording, it can be time consuming or costly to obtain such data. In addition, to simulate context data during data-driven tests, testers have to correctly configure simulators and applications in a proper environment. Complicated modification of OS or applications is sometimes needed. For model-based approaches, testers have to define an abstract model to represent the application. Despite a few automatic modelling techniques, most

model-based approaches require testers to manually input a specialised model. Compared to data-driven approaches, most model-based approaches are based on common software or hardware dependencies. Only few model-based approaches involve complicated configuration.

5.2 Special- and General-Purpose Context Simulation

Regardless of being data driven or model based, all context simulation approaches can be grouped into two categories: special-purpose and general-purpose approaches.

5.2.1 Special-Purpose Context Simulation. Special-purpose approaches focus only on one type of mobile context-aware applications. A typical example is location simulation for location-based services. Beyond other simulation approaches only considering location coordinates, location simulators take into account more geographical characteristics such as buildings, roads, and traffic. The actions of location-based services may vary in different geographical conditions. For example, vehicle navigation applications may use road and traffic information to avoid traffic jams. Some location simulators (e.g., CATLES [30]) even provide a virtual 3D environment as a control interface for testers. Special-purpose context simulation also include multimedia simulators (e.g., CamTest [71]) that can simulate audio or video streams to test mobile applications based on microphone or camera. Hence, special-purpose approaches can effectively examine the features of specific mobile context-aware applications.

5.2.2 General-Purpose Context Simulation. General-purpose approaches aim at common features of mobile context-aware applications. Since a large number of mobile context-aware applications take into account network conditions and sensor readings, most general-purpose context simulators support the simulation of network status and sensor data collection. In addition general-purpose context simulators accept applications with an arbitrary set of different context events. Testers can select a suitable combination of context events for an application in simulations. Some general-purpose context simulators can even summarise and prioritise context situations that are likely to cause problems in different kinds of applications. Thus, general-purpose context simulation is suitable to examine the features that are in common with regular mobile context-aware applications.

5.3 The Notion of Context in Simulation

In the field of mobile computing, the notion of context is essentially open ended in different scenarios. Likewise, the connotation of context varies among simulation approaches.

5.3.1 Physical and Virtual Context. Most context simulation approaches focus only on physical context such as location, network status, and sensor readings, since many mobile context-aware applications only take physical context as input. For example, location simulators produce dummy location information for location-based services. As well, many context simulators are able to generate sensor readings, such as acceleration and luminosity, for applications that track human/object physical activities and ambient environments. In addition to physical context, a small number of context simulation approaches support the simulation of virtual context, such as smartphone notifications, application output/crashes, and user input text. The simulation of virtual context can benefit the validation of applications that rely on software information and human input.

5.3.2 General and Personalised Context. The majority of context simulation approaches do not consider individual differences of users'/objects' context. These approaches use general context as test cases to represent the common characteristics of a group of users or objects. In contrast, several context simulation approaches can distinguish personalised context patterns from individual users/objects. This function is necessary for testing applications that should adjust decisions for

Table 13. Summary of Challenges and Future Opportunities

Challenge	Desired Feature	Supportive Technique
Early-stage testing	Continuous sensing simulation	Combinatorial optimisation
	Context model evaluation	Model visualisation
Emulation fidelity	High-fidelity execution	
	High-fidelity injection	
	Device fragmentation	Mobile virtualisation
Context heterogeneity	Multi-channel context replay	Multimedia synchronisation
	Simulation of external sensors	
Multiple devices	Simulation of multi-device sensing	Multi-device synchronisation
Automation support	Auto model extraction	NLP
	Auto model transformation	Quiver mutation, deep learning
	Auto test configuration	Cloud testing, cross compiling

different user demographics. For instance, age and gender are importance factors for judging the correct output of healthcare-related mobile context-aware applications.

5.3.3 Context Granularity. The granularity of context differs across context simulation approaches. Some approaches use coarse-level context. For example, to test applications that rely on WiFi, simulators may only generate the on/off state of WiFi. Although this granularity can test features such as data synchronisation, it cannot provide WiFi Received Signal Strength Information (RSSI) of an access point to test features such as indoor positioning and activity recognition. Hence, some approaches support the simulation of fine-grained context that includes more low-level attributes related to context events.

5.4 Challenges and Future Opportunities

Despite the advances in context simulation techniques for testing mobile context-aware applications, there are still several unexplored issues in this field. Hence, we propose a vision of existing challenges and desired features of future context simulation approaches. Specifically, we discuss how current techniques that have not been applied in this field can support future context simulation. Finally, we provide some deliberations about potential architectures of future context simulators.

Table 13 shows a summary of challenges and future opportunities, including desired features of future context simulation approaches, and current techniques that have not been applied in this field can support future context simulation.

5.4.1 Context Simulation in Early-Stage Testing. The current trend for context simulation is to test implemented mobile context-aware applications. Only a handful of model-based approaches aim to perform context simulation before the implementation stage. In addition to software bugs introduced in coding activities, software flaws can emerge from improper requirement analysis and software design [54]. Early-stage testing can reduce the difficulty and cost of software repair [97].

Hence, future context simulation approaches can place more emphasis on requirement analysis and software design of mobile context-aware applications. A desired features is the validation of continuous sensing design which often contains tradeoffs between performance and overhead [59]. Developers can use this feature to make better choices of sensors and duty cycling algorithms during the design phase, avoiding time-consuming trial and error in experiments on different implemented versions. Combinatorial optimisation is a promising supportive technique to find the optimal design choice rather than an exhaustive search. In addition, at the very early stage of

development, conducting context simulation can help developers design and evaluate context models [14]. In this sense, a desired feature is the evaluation of context models via early-stage context simulation. Context simulators can generate various context events to preliminarily assess context models. Visualising the change of model states is a supportive technique for developers to understand and improve their context models.

5.4.2 Emulation Fidelity. Although context simulation on physical devices can obtain reliable results in some types of tests (e.g., real timeness in performance testing), emulation on PCs or the cloud is more suitable for low-budget and large-scale tests.

Current emulators (e.g., Genymotion and BlueStacks) are designed to maximise their computation speed and minimise computation cost [55]. The major goals of these emulators include (1) playing mobile video games on PCs and (2) offloading computation-intensive tasks from mobile devices to the cloud. Although these emulators can obtain good performance in computation tasks and may be used for functional tests of mobile applications, they cannot reflect how well applications perform on physical devices. Hence, testers cannot use these emulators to examine non-functional properties of applications. To avoid significant performance differences between test results on emulators and physical devices, future research can explore new techniques that enable high-fidelity emulators. A desired feature for future emulators is to reflect actual time and resource cost of executions on various mobile platforms, because PCs or cloud servers have more powerful hardware (e.g., processors) than physical devices [116]. Meanwhile, considering that mobile context-aware applications are supposed to collect data from physical device sensors, another desired feature for future emulators is to equalise their computational cost of context simulation and the expected computational cost of real-world data collection on physical devices.

In addition, since mobile applications may perform differently on various device models and OS [18, 70], a desired feature for future emulators is to realise device fragmentation caused by different characteristics of physical devices. Advanced mobile virtualisation techniques can be helpful for future emulators to achieve this feature on regular PCs or cloud servers.

5.4.3 Context Heterogeneity. As mobile devices tend to contain a wider range of sensors and modules, the increasing diversity of contextual data is a great challenge in context simulation.

Future research needs to develop new tools that can provide sufficient coverage of such heterogeneous contextual data. For instance, considering that phone-based car driving monitoring applications (e.g., You et al. [117]) may use dual cameras, a microphone, and inertial sensors, a desired feature for future context simulation is multi-channel context replay, including multi-channel video frames, audio streams, and sensor events. Multimedia synchronisation is a necessary technique to support this feature by delivering each type of context data at the correct timing. Further, some mobile devices are able to collect data from external sensors, including USB accessories (e.g., FLIR One Thermal Imaging Camera Attachment [26]) and wireless sensors. A desired feature of future context simulators is to perform context simulation for applications relying on external sensors.

5.4.4 Multiple Devices. Multi-device context-aware applications can bring a few new issues to context simulation [74]. Existing methods have limited features to simulate context scenarios where various devices collaborate to sense the environment. In future context simulators, a desired feature is to simulate the communication and collaboration within a certain number of mobile devices serving a context-aware system. A typical example is the personal area network [108] where personal devices, such as smartphones and smartwatches, are connected. It is worth noting that many applications for wearable devices contain some components operating on smartphones. Smart homes with diverse intelligent appliances is also an important area for further investigation.

To achieve the context simulation for multi-device scenarios, a supportive technique is to synchronise various context events from sensors on each device with respective timestamps to validate the entire context-aware system across devices. In addition, to simulate multiple components of an application, the simulator should be able to monitor and manage the components across devices or emulators. Suitable signalling mechanisms are needed for the cross-component communications such as sending commands, delivering context data, and reporting test results.

5.4.5 Automation Support. Some existing approaches require testers to manually complete specific operations (e.g., providing human-generated software models), lacking automation support for context simulation. Hence, testers must have sufficient knowledge to apply these approaches.

To lower this barrier, future tools should provide automation features. For example, a desired feature is automatic model extraction that can produce software models by parsing the source code of applications that are written in various programming languages and for different OS. NLP can be a supportive technique for future tools to understand semantics of context collection and processing from software source code, similar to the automatic test case generation mechanism in Snowdrop [122]. As testers often have software design models at the software design stage, a desired feature is automatic model transformation that can convert these models for model-based context simulation approaches, such as converting UML diagrams into different types of finite state machines. To achieve such a feature, quiver mutation [111] (analysing and transforming directed graphs using algebra) and deep learning can serve as supportive techniques to analyse and generate equivalent models in different forms.

Another desired feature is to automatically adjust test configuration on various execution environments of hardware, since testers may conduct tests on different hardware and OS that may cause compatibility issues, such as inconsistent data formats and debugging features. In particular, cloud testing can be a supportive technique to automatically customise execution environments (e.g., modifying standard OS SDK) for testers so that testers can avoid the pain of flashing OS images on their own devices. With a large number of physical devices hosted on cloud platforms, a cross-compiling technique is also necessary for testers to quickly deploy appropriate versions of their applications in each customised execution environment.

5.4.6 Potential Architectures of Context Simulators. Architectures of current context simulation approaches are seldom mentioned in the literature. Thus, there is a lack of standards about what components they should have. In the future, it is likely that context simulators will be built in a modular fashion, as they have multiple features for different stages.

Future context simulators may have both data-driven and model-based context simulation to support the validation of mobile context-aware applications in the entire software development life cycle. During the design phase, model-based context simulation can help developers examine their design choices. After implementation, data-driven context simulation can validate the finished software products. For model-based context simulation, future context simulators should contain model management modules for the definition, modification, extraction, and transformation of application models. Model-based context simulators should also contain simulation management modules for the configuration, test case generation, execution, and summarisation of tests. Likewise, future data-driven context simulators should contain data management modules to record, generate, or manipulate test data. In addition, future data-driven context simulators should contain simulation management modules for the configuration, test data import, execution, and summarisation of tests. If context simulation approaches require the modification of OS or applications, they should also specific modules for OS and app modification. As well, cloud-based context simulation approaches should contain remote control modules as an interface for testers to control simulation processes, reducing the cost of simulation and the efforts of testers.

6 CONCLUSION

This article presents a survey of context simulation techniques for testing mobile context-aware applications. We first characterise modern mobile devices, sensors, and mobile context-aware applications. Then, we identify the need of context simulation techniques for testing mobile context-aware applications. With regard to the need, we provide an in-depth discussion and comparison of the key technical details of relevant context simulation techniques including both data-driven and model-based approaches. At the end of this survey, we highlight several unexplored issues and future directions for further advancements in this field.

Testing mobile context-aware applications is often a costly and time-consuming process. The emergence of context simulation techniques alleviates this problem by replacing impractical real-world experiments with efficient simulated executions. However, context simulation on mobile context-aware applications is a technically challenging task. Despite existing approaches presented in the literature, more research is needed to help testers in a wide range of testing activities and scenarios for mobile context-aware applications. In addition, further study on these approaches is needed to help researchers find knowledge gaps and inspirations in the literature, for example, by technical reports and systematic mapping on related techniques, tools, and publications.

REFERENCES

[1] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. IEEE, Los Alamitos, CA, 1016–1026.

[2] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, New York, NY, 83–93.

[3] Wi-Fi Alliance. 2019. Wi-Fi Direct. Retrieved January 4, 2020 from https://www.wi-fi.org/discover-wi-fi/wi-fi-direct.

[4] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Nicola Amatucci. 2013. Considering context events in event-based testing of mobile applications. In *Proceedings of the IEEE 6th International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'13)*. IEEE, Los Alamitos, CA, 126–133.

[5] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, 258–261.

[6] Amit Seal Ami, Md. Mehedi Hasan, Md. Rayhanur Rahman, and Kazi Sakib. 2018. MobiCoMonkey: Context testing of Android apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft'18)*. ACM, New York, NY, 76–79.

[7] Apple. 2019. Xcode 11. Retrieved January 4, 2020 from https://developer.apple.com/xcode/.

[8] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. 2018. Towards model checking Android applications. *IEEE Transactions on Software Engineering* 44, 6 (2018), 595–612.

[9] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. 2007. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing* 2, 4 (2007), 263–277.

[10] Gregory Biegel and Vinny Cahill. 2004. A framework for developing mobile, context-aware applications. In *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications (PerCom'04)*. IEEE, Los Alamitos, CA, 361–365.

[11] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable race detection for android applications. *ACM SIGPLAN Notices* 50 (2015), 332–348.

[12] Jiang Bo, Long Xiang, and Gao Xiaopeng. 2007. MobileTest: A tool supporting automatic black box test for software on smart mobile devices. In *Proceedings of the 2nd International Workshop on Automation of Software Test*. IEEE, Los Alamitos, CA, 8.

[13] Szymon Bobek. 2016. Context Simulator (KnowMe). Retrieved January 4, 2020 from http://glados.kis.agh.edu.pl/.

[14] Szymon Bobek, Sebastian Dziadzio, Paweł Jaciów, Mateusz Ślażyński, and Grzegorz J. Nalepa. 2015. Understanding context with context viewer–tool for visualization and initial preprocessing of mobile sensors data. In *Proceedings of the International and Interdisciplinary Conference on Modeling and Using Context*. 77–90.

[15] Calabash. 2019. Calabash-Android. Retrieved January 4, 2020 from https://github.com/calabash/calabash-android.
[16] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2004. Adaptive random testing. In *Proceedings of the Annual Asian Computing Science Conference*. 320–329.
[17] Kwang-Ting Cheng and Avinash S. Krishnakumar. 1993. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 1993 30th Conference on Design Automation*. IEEE, Los Alamitos, CA, 86–91.
[18] Min Choi and Seung-Ho Lim. 2016. x86-Android performance improvement for x86 smart mobile devices. *Concurrency and Computation: Practice and Experience* 28, 10 (2016), 2770–2780.
[19] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet?. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, Los Alamitos, CA, 429–440.
[20] Progress Software Corporation. 2019. Fiddler. Retrieved January 4, 2020 from https://www.telerik.com/fiddler.
[21] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (1995), 273–297.
[22] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the 12th European Conference on Computer Systems*. ACM, New York, NY, 145–160.
[23] Dex2jar Group. 2019. Dex2jar. Retrieved January 4, 2020 from https://github.com/pxb1988/dex2jar/.
[24] David Erickson, Dakota O'Dell, Li Jiang, Vlad Oncescu, Abdurrahman Gumus, Seoho Lee, Matthew Mancuso, and Saurabh Mehta. 2014. Smartphone technology can be transformative to the deployment of lab-on-chip diagnostics. *Lab on a Chip* 14, 17 (2014), 3159–3164.
[25] Denzil Ferreira, Vassilis Kostakos, and Anind K. Dey. 2015. AWARE: Mobile context instrumentation framework. *Frontiers in ICT* 2 (2015), 6.
[26] FLIR. 2019. FLIR ONE Gen 3 Thermal Camera for Smart Phones. Retrieved January 4, 2020 from https://www.flir.com/products/flir-one-gen-3/.
[27] Huber Flores, Denzil Ferreira, Chu Luo, Vassilis Kostakos, Pan Hui, Rajesh Sharma, Sasu Tarkoma, and Yong Li. 2016. Social-aware device-to-device communication: A contribution for edge and fog computing? In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM, New York, NY, 1466–1471.
[28] Eclipse Foundation. 2019. Eclipse. Retrieved January 4, 2020 from https://www.eclipse.org/.
[29] Jerry Gao, Xiaoying Bai, Wei-Tek Tsai, and Tadahiro Uehara. 2014. Mobile application testing: A tutorial. *Computer* 2 (2014), 46–55.
[30] Sandro Rodriguez Garzon, Bersant Deva, Benoît Hanotte, and Axel Küpper. 2016. CATLES: A crowdsensing-supported interactive world-scale environment simulator for context-aware systems. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, New York, NY, 77–87.
[31] Sandro Rodriguez Garzon and Dmytro Hritsevskyy. 2012. Model-based generation of scenario-specific event sequences for the simulation of recurrent user behavior within context-aware applications (WIP). In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*. 29.
[32] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, New York, NY, 103–116.
[33] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 72–81.
[34] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *Proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)*. IEEE, Los Alamitos, CA, 88–99.
[35] Google. 2019. Android Debug Bridge (adb). Retrieved January 4, 2020 from https://developer.android.com/studio/command-line/adb.
[36] Google. 2019. Android Getevent. Retrieved January 4, 2020 from https://source.android.com/devices/input/getevent.
[37] Google. 2019. Android App Manifest Overview. Retrieved January 4, 2020 from https://developer.android.com/guide/topics/manifest/manifest-intro.
[38] Google. 2019. Android Studio. Retrieved January 4, 2020 from https://developer.android.com/studio.
[39] Google. 2019. Espresso. Retrieved January 4, 2020 from https://developer.android.google.cn/reference/android/support/test/espresso/Espresso.
[40] Google. 2019. Firebase Test Lab. Retrieved January 4, 2020 from https://firebase.google.com/docs/test-lab.
[41] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. 2003. An introduction to the testing and test control notation (TTCN-3). *Computer Networks* 42, 3 (2003), 375–403.

[42] 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*. ACM, New York, NY, 420–427.
[43] Tobias Griebe, Marc Hesenius, and Volker Gruhn. 2015. Towards automated UI-tests for sensor-based mobile applications. In *Proceedings of the International Conference on Intelligent Software Methodologies, Tools, and Techniques*. 3–17.
[44] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, 204–217.
[45] Klaus Havelund and Thomas Pressburger. 2000. Model checking Java programs using Java Pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381.
[46] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 2013. Slicing droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, New York, NY, 1844–1851.
[47] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295.
[48] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, New York, NY, 18.
[49] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for Android. *ACM SIGPLAN Notices* 50 (2015), 349–366.
[50] Yongjian Hu and Iulian Neamtiu. 2016. Fuzzy and cross-app replay for smartphone apps. In *Proceedings of the IEEE/ACM 11th International Workshop in Automation of Software Test (AST'16)*. IEEE, Los Alamitos, CA, 50–56.
[51] Yongjian Hu and Iulian Neamtiu. 2016. VALERA: An effective and efficient record-and-replay tool for Android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, New York, NY, 285–286.
[52] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically verifying and reproducing event-based races in Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, New York, NY, 377–388.
[53] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. 2017. Chizpurfle: A gray-box Android fuzzer for vendor service customizations. In *Proceedings of the IEEE 28th International Symposium on Software Reliability Engineering (ISSRE'17)*. IEEE, Los Alamitos, CA, 1–11.
[54] Pankaj Jalote. 2008. *A Concise Introduction to Software Engineering*. Springer Science & Business Media.
[55] Padmaja Joshi, Ashwin Nivangune, Ranjan Kumar, Sathish Kumar, Rakesh Ramesh, Sushant Pani, and Arif Chesum. 2015. Understanding the challenges in mobile computation offloading to cloud through experimentation. In *Proceedings of the 2015 2nd ACM International Conference on Mobile Software Engineering and Systems*. IEEE, Los Alamitos, CA, 158–159.
[56] Milan Jovic, Andrea Adamoli, Dmitrijs Zaparanuks, and Matthias Hauswirth. 2010. Automating performance testing of interactive Java applications. In *Proceedings of the 5th Workshop on Automation of Software Test*. ACM, New York, NY, 8–15.
[57] Ernesto Kofman and R. D. Castro. 2006. STDEVS, a novel formalism for modeling and simulation of stochastic discrete event systems. In *Proceedings of the 2006 AADECA Conference*.
[58] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2018. Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability* 99 (2018), 1–22.
[59] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. 2010. A survey of mobile phone sensing. *IEEE Communications Magazine* 48, 9 (2010), 140–150.
[60] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10 (1966), 707–710.
[61] Chieh-Jan Mike Liang, Nic Lane, Niels Brouwers, Li Zhang, Börje Karlsson, Ranveer Chandra, and Feng Zhao. 2013. *Contextual Fuzzing: Automated Mobile App Testing Under Dynamic Device and Environment Conditions*. Microsoft.
[62] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, et al. 2014. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*. ACM, New York, NY, 519–530.
[63] Libuv Team. 2019. Libuv. http://libuv.org/.
[64] Santiago Liñán, Laura Bello-Jiménez, María Arévalo, and Mario Linares-Vásquez. 2018. Automated extraction of augmented models for Android apps. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*. IEEE, Los Alamitos, CA, 549–553.
[65] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, Los Alamitos, CA, 399–410.

[66] Hongbo Liu, Jie Yang, Simon Sidhom, Yan Wang, Yingying Chen, and Fan Ye. 2014. Accurate WiFi based localization for smartphones using peer assistance. *IEEE Transactions on Mobile Computing* 13, 10 (2014), 2199–2214.

[67] Zhifang Liu, Xiaopeng Gao, and Xiang Long. 2010. Adaptive random testing of mobile application. In *Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET'10)*, Vol. 2. IEEE, Los Alamitos, CA.

[68] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. 2009. SoundSense: Scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, 165–178.

[69] 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 *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM, New York, NY, 1448–1453.

[70] 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. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 80.

[71] Chu Luo, Zewen Xu, Ruining Dong, Jorge Goncalves, Eduardo Velloso, and Vassilis Kostakos. 2019. CamTest: A laboratory testbed for camera-based mobile sensing applications. *Pervasive and Mobile Computing* 56 (2019), 106–131.

[72] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, 224–234.

[73] Tim A. Majchrzak and Matthias Schulte. 2015. Context-dependent testing of applications for mobile devices. *Open Journal of Web Technologies* 2, 1 (2015), 27–39.

[74] Lakhdar Meftah, Maria Gomez, Romain Rouvoy, and Isabelle Chrisment. 2017. ANDROFLEET: Testing WiFi peer-to-peer mobile apps in the large. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA, 961–966.

[75] Mirza Aamir Mehmood, M. N. A. Khan, and Wasif Afzal. 2018. Automating test data generation for testing context-aware applications. In *Proceedings of the 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS'18)*. IEEE, Los Alamitos, CA, 104–108.

[76] Zhanshuai Meng, Yanyan Jiang, and Chang Xu. 2015. Facilitating reusable and scalable automated testing and analysis for Android apps. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*. ACM, New York, NY, 166–175.

[77] Microsoft. 2019. Xamarin Test Cloud. Retrieved January 4, 2020 from https://testcloud.xamarin.com/.

[78] Zoltán Micskei, Zoltán Szatmári, János Oláh, and István Majzik. 2012. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In *Proceedings of the KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*. 504–513.

[79] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*. 3111–3119.

[80] Robin Milner. 2006. Pure bigraphs: Structure and dynamics. *Information and Computation* 204, 1 (2006), 60–122.

[81] Kevin Moran, Richard Bonett, Carlos Bernal-Cárdenas, Brendan Otten, Daniel Park, and Denys Poshyvanyk. 2017. On-device bug reporting for Android applications. In *Proceedings of the IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft'17)*. IEEE, Los Alamitos, CA, 215–216.

[82] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. CrashScope: A practical tool for automated testing of Android applications. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C'17)*. IEEE, Los Alamitos, CA, 15–18.

[83] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE, Los Alamitos, CA, 29–35.

[84] Damien Octeau, Somesh Jha, and Patrick McDaniel. 2012. Retargeting Android applications to Java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 6.

[85] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. 2010. Paranoid Android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, New York, NY, 347–356.

[86] Yi Qin, Chang Xu, Ping Yu, and Jian Lu. 2016. SIT: Sampling-based interactive testing for self-adaptive apps. *Journal of Systems and Software* 120 (2016), 70–88.

[87] 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, New York, NY, 571–582.

[88] Andres J. Ramirez, Adam C. Jensen, and Betty H. C. Cheng. 2012. A taxonomy of uncertainty for dynamically adaptive systems. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, Los Alamitos, CA, 99–108.
[89] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. 2014. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, 190–203.
[90] Renas Reda and Hugo Josefson. 2019. Robotium. https://github.com/RobotiumTech/robotium.
[91] Luigi Rizzo. 1997. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review* 27, 1 (1997), 31–41.
[92] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *The Unified Modeling Language Reference Manual*. Pearson Higher Education.
[93] Michele Sama and David S. Rosenblum. 2019. TestingEmulator. Retrieved January 4, 2020 from https://code.google.com/archive/p/testingemulator/.
[94] Michele Sama, David S. Rosenblum, Zhimin Wang, and Sebastian Elbaum. 2008. Model-based fault detection in context-aware adaptive applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, 261–271.
[95] Sakura She, Sasindran Sivapalan, and Ian Warren. 2009. Hermes: A tool for testing mobile device applications. In *Proceedings of the Australian Software Engineering Conference (ASWEC'09)*. IEEE, Los Alamitos, CA, 121–130.
[96] Open Signal. 2019. Open Signal Reports. Retrieved January 4, 2020 from http://opensignal.com.
[97] Yogesh Singh. 2012. *Software Testing*. Cambridge University Press, Cambridge, England.
[98] Kwangsik Song, Ah-Rim Han, Sehun Jeong, and Sung Deok Cha. 2015. Generating various contexts from permissions for testing Android applications. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE'15)*. 87–92.
[99] Oleksii Starov, Sergiy Vilkomir, Anatoliy Gorbenko, and Vyacheslav Kharchenko. 2015. Testing-as-a-service for mobile applications: State-of-the-art survey. In *Dependability Problems of Complex Information Systems*. Springer, 55–71.
[100] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. 2016. Generating test cases to expose concurrency bugs in Android applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, 648–653.
[101] Testing Technologies. 2015. TTworkbench. https://www.spirent.com/products/ttworkbench.
[102] Ralf Tönjes, Eike Steffen Reetz, Marten Fischer, and Daniel Kuemper. 2015. Automated testing of context-aware applications. In *Proceedings of the 2015 IEEE 82nd Vehicular Technology Conference (VTC2015-Fall)*. 1–5.
[103] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2018. Automated functional testing of mobile applications: A systematic mapping study. *Software Quality Journal* 27, 1 (2018), 149–201.
[104] Connor Tumbleson and Ryszard Wiśniewski. 2019. Apktool. Retrieved January 4, 2020 from https://ibotpeaches.github.io/Apktool/.
[105] Uber. 2019. Uber—Earn Money by Driving or Get a Ride Now. Retrieved January 4, 2020 from https://www.uber.com.
[106] Asmau Usman, Noraini Ibrahim, and Ibrahim Anka Salihu. 2018. Test case generation from Android mobile applications focusing on context events. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*. ACM, New York, NY, 25–30.
[107] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.
[108] Changhong Wang, Wei Lu, Michael R. Narayanan, Stephen J. Redmond, and Nigel H. Lovell. 2015. Low-power technologies for wearable telecare and telehealth systems: A review. *Biomedical Engineering Letters* 5, 1 (2015), 1–9.
[109] Yichen Wang and Yikun Wang. 2011. Model-based simulation testing for embedded software. In *Proceedings of the 3rd International Conference on Communications and Mobile Computing (CMC'11)*. IEEE, Los Alamitos, CA, 103–109.
[110] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. 2007. Automated generation of context-aware tests. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 406–415.
[111] Matthias Warkentin. 2014. *Exchange Graphs via Quiver Mutation*. Ph.D. Dissertation. Technische Universitat Chemnitz.
[112] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 439–449.
[113] Chang Xu, S. C. Cheung, Wing Kwong Chan, and Chunyang Ye. 2010. Partial constraint checking for context consistency in pervasive computing. *ACM Transactions on Software Engineering and Methodology* 19, 3 (2010), 9.
[114] Chang Xu, Shing-Chi Cheung, Xiaoxing Ma, Chun Cao, and Jian Lu. 2012. Dynamic fault detection in context-aware adaptation. In *Proceedings of the 4th Asia-Pacific Symposium on Internetware*. ACM, New York, NY, 1.

[115] Edgardo Barsallo Yi, Amiya Maji, and Saurabh Bagchi. 2018. How reliable is my wearable: A fuzz testing-based study. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*. IEEE, Los Alamitos, CA, 410–417.

[116] Tetsuya Yoshida, Hiroshi Yamada, and Kenji Kono. 2009. Using a virtual machine monitor to slow down CPU speed for embedded time-sensitive software testing. *Information and Media Technologies* 4, 4 (2009), 870–884.

[117] Chuang-Wen You, Martha Montes-de Oca, Thomas J. Bao, Nicholas D. Lane, Hong Lu, Giuseppe Cardone, Lorenzo Torresani, and Andrew T. Campbell. 2012. CarSafe: A driver safety app that detects dangerous driving behavior using dual-cameras on smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM, New York, NY, 671–672.

[118] Lian Yu, Wei Tek Tsai, Yanbing Jiang, and Jerry Gao. 2014. Generating test cases for context-aware applications using bigraphs. In *Proceedings of the 8th International Conference on Software Security and Reliability (SERE'14)*. IEEE, Los Alamitos, CA, 137–146.

[119] Xingzi Yuan, Omid Setayeshfar, Hongfei Yan, Pranav Panage, Xuetao Wei, and Kyu Hyung Lee. 2017. DroidForensics: Accurate reconstruction of Android attacks via multi-layer forensic logging. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, New York, NY, 666–677.

[120] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. 2000. *Theory of Modeling and Simulation*. Academic Press.

[121] Samer Zein, Norsaremah Salleh, and John Grundy. 2016. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software* 117 (2016), 334–356.

[122] Li Lyna Zhang, Chieh-Jan Mike Liang, Yunxin Liu, and Enhong Chen. 2017. Systematically testing background services of mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, Los Alamitos, CA, 4–15.

[123] Li Lyna Zhang, Chieh-Jan Mike Liang, Wei Zhang, and Enhong Chen. 2017. Towards a contextual and scalable automated-testing service for mobile apps. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*. ACM, New York, NY, 97–102.

[124] Oum-El-Kheir Aktouf, Tao Zhang, Jerry Gao, and Tadahiro Uehara. 2015. Testing location-based function services for mobile applications. In *Proceedings of the IEEE Symposium on Service-Oriented System Engineering (SOSE'15)*. IEEE, Los Alamitos, CA, 308–314.

Received May 2019; revised September 2019; accepted November 2019