Testing Context-Aware Services based on Smartphones by Agent Based Social Simulation

Pablo Campillo-Sanchez a,∗, Emilio Serrano b and Juan A. Botía a

a Information Engineering and Communications Department, University of Murcia, 30100, Murcia, Spain
E-mail: {pablocampillo, juanbot}@um.es
b Telematic Systems Engineering Department, Polytechnic University of Madrid, 28040, Madrid, Spain
E-mail: eserrano@gsi.dit.upm.es

Abstract. Smartphones are increasingly popular each year and have more and better sensors. These sensors are a rich information source for creating context-aware applications. Testing these applications directly in living labs is very expensive since it involves, among other things, setting up a reproduction of the environment where the application or service under test will be executed. This paper proposes the UbikMobile architecture which aims to provide developers with tools to test context-aware services based on smartphones in a simulated environment. The primary goal in UbikMobile is to reduce the faults in the software as much as possible. The main contribution is that a real smartphone, and not only an emulator, can be connected to the simulated world to conduct: (1) manual tests (i.e. the developer controls an avatar in the simulation); and, (2) automatic tests (i.e. agents in the simulation are capable of interacting automatically with the smartphone). This paper shows the main design decisions in the architecture presented to allow the interested reader to reproduce or to extend our results. It also includes a number of case studies to illustrate how to use the architecture.

Keywords: Mobile devices, Android, Context-aware services, Testing, Agent based social simulation

1. Introduction

Smartphones are equipped with sensors, such as GPS, accelerometer, gyroscope, camera and microphone. Such heterogeneous hardware allow smartphones to gather context information about the user, such as date and time, location and activity. Context is used to develop context-aware applications, offering services to users depending on their needs. Despite the promising potential of using smartphones as context information sources, there are technical issues that need to be solved. One of these issues in context-aware application development is software testing [30].

Testing software is the process of executing a program in order to find errors in the code [7]. Errors must then be debugged. The main challenge of testing context aware applications is to generate a set of values for each sensor, useful and meaningful for each test. This challenge is even bigger when some of the sensors must show correlated values.

The most extended approach for testing context-aware applications is by using Living Labs. These laboratories allow users and researchers to work together in a real life environments to evaluate the quality of the context-service developed. Although this testing approach is intuitive and, ultimately, necessary, it is possible to anticipate a large number of faults before software deployment using simulation. This anticipation involves a significant cost reduction in fixing these faults because one of the bases of software engineering is that the earlier a fault is found, the cheaper it is to fix it [20] and applying well-established software engineering principles is fundamental in context-aware services [27] such as date and time, location and activ-
Let us suppose a service, executed at the smartphone, that offers indoor location information such as the one explained in [12]. It uses the digital compass to record orientation changes and the accelerometer to infer movement. To test this service, we could generate several tests defined by two time series: accelerations for accelerometer and radians for digital compass. Given an initial location and these time series, the service estimates a new location. How to generate the time series of the example? The approach of this paper is based on simulating humans and sensors within an artificial environment. Simulated humans move through the artificial environment, carrying the smartphone with them. And the simulated sensors generate the corresponding sensor data. Thus, the definition of a test is merely defining what the sequence of actions of the simulated human. To define tests in this way is much more natural, comprehensible and realistic.

The contribution of this paper is the UbikMobile architecture. It is an architecture for testing context-aware services and applications running on smartphones. Software tests are generated by means of agent based social simulation (ABSS). UbikMobile allows real smartphones\(^1\) be connected to a simulated world. In this way, real software is tested with simulated situations of interest. In this case, the System Under Test (SUT) is either an application or service running in the smartphone.

In the architecture, tests may be conducted manually by the developer playing the role of SUT’s user. An alternative manner is defining realistic users by means of simulation models and replace users of context-aware applications [10]. This second manner produces totally automatic software tests. This combination of real and simulated elements defines tests as a series of concrete situations and user behaviours in the virtual world. Testing is not only more natural and realistic, but also more understandable, and therefore, more robust and extendable. UbikMobile is an extension of UbikSim\(^2\), designed to simulate environments, domotic devices and users interacting with real ubiquitous computing systems.

There are other simulators that connect with real smartphones. However, to the best of our knowledge, UbikMobile is the first to allow a simulation capable of interacting automatically with a real smartphone. This is important to provide developers with automatic testing of context-aware services. Currently, UbikMobile only deals with smartphone applications for Android operating system\(^3\). Android is open source and the approach presented in this paper needs to minimally modify the operating system source code (see section 5 for details). Adapting UbikMobile to Apple’s iOS is a pending work since it counts with a very important community of users.

The remainder of the paper is structured as follows. After reviewing the background in section 2, an overview of the architecture proposed is offered in section 3. Section 4 introduced how to design simulated worlds for software testing. Section 5 presents how UbikMobile enables connecting a real smartphone with such simulated worlds. Section 6 illustrates how to model users and their interactions by ABSS for automatic testing. Section 7 presents three different case studies. Section 8 analyses the related works and section 9 concludes.

2. Background

The most common Android development tools [21] are composed of SDK (Software Development Kit) and ADT plug-in (Android Development Tools), both of which are open source. The SDK includes the Android APIs (Application Programming Interface), development tools and the Android Virtual Device Manager and Emulator. ADT plugin extends the SDK functionalities to an IDE (Integrated Development Environment).

The Android software stack comprises several layers. The Application Framework Layer is the most significant here. It provides (1) classes used to create Android applications, (2) a generic abstraction for access to the hardware (e.g. sensors) and (3) it manages the user interface and application resources. The following subsections refer to some Android services related to environment sensing, located in this layer. They are of interest in this paper because they are either modi-

---

\(^1\)Hereinafter, the term smartphone is used to refer to any device with Android OS such as tablets and emulators of smartphones.


\(^3\)Android website: [http://www.android.com/](http://www.android.com/)
fied in some aspect or directly used by the UbikMobile architecture (see section 5 for details. The subsections also include an accounting of existing testing tools that help manage such services with the purpose of making software tests.

2.1. Location issues in Android

Android location services are mainly obtained through CellID or GPS services. GPS is the most extended method because of its better accuracy. On the other hand, this method only works in outdoor environments. CellID approximates location based on the urban cell where the smartphone is located at. It gets lower accuracy but works indoors. These services are accessed through the LocationManager class.

With respect to useful tools for testing using such services, an interesting Android tool called Dalvik Debug Monitor Server (DDMS) allows the programmer to simulate different locations that a user passes through without moving the smartphone. Tests using such locations in software tests is actually not realistic. Locations are written down to a file as a list of coordinates. The simulation reproduces constant velocity and straight line displacements. More realism could be reached if the coordinates were given by simulated person displacements in the user’s natural environment. Furthermore, a location-based service based also on inertial devices could not be tested with the help of this tool.

2.2. Sensor issues in Android

Sensors on Android are managed in a similar way to location. The SensorManager class gives access to all the sensors in the smartphone. Obviously, sensors available at the smartphone to be used by the SUT depend on the specific model.

Some third-party tools help working with Android sensors. For example, Sensor Simulator is a stand-alone application of OpenIntents. It allows developers to simulate movement of the smartphone and its sensors merely by moving the mouse.

Another tool example is Samsung Sensor Simulator. This tool is a user interface with a model of a smartphone. By manipulating the position of the device or the forces acting on the device, new sensor readings can be generated, under a variety of conditions. It also allows to connect to a real device and gather real sensor readings. Using these tools for software testing is not an easy task. It is not easy to generate with them time series of sensor data corresponding to concrete situations of interest (e.g. the user starts running and suddenly he/she falls).

2.3. Audio and Video issues in Android

Audio and video are also important sources for context sensing. They are interesting because of their processing, e.g. image processing to detect objects and audio processing to recognize speech commands, for example. Android offers these services through Camera and AudioRecord classes for video and audio processing, respectively.

Regarding available tools to work with Android audio and video, Android Emulator allows using a webcam and the PC integrated microphone as sources of video and audio respectively. However, the emulator is not specially prepared for testing a SUT. Thus, it can be said that regarding audio and video, there is a clear lack of tools for software testing.

3. UbikMobile architecture

This section aims presents a platform which allows Android applications to be tested by simulation, UbikMobile. This testing can be of two types: manual or automatic. Manual testing is carried out by the developer who is testing the SUT installed on an Android device. Such device is connected to its counterpart in the simulator. Testing is done by the developer playing the role of the user, controlling the avatar with the keyboard. The developer interacts both with the avatar and with the real device. In contrast, automatic testing is carried out by agents simulating users.

Manual testing requires a means to make a real smartphone sense the simulated world as real (e.g. if the avatar uses the smartphone’s camera, the video generated by the simulated sensor should be received and displayed at the real smartphone as it were generated by the real sensor) [9]. Besides, automatic testing needs that agents interact with the real device (e.g. to launch applications, to press buttons, etc.). This involves getting feedback from the smartphone to allow agents to know its state. Both facilities are offered by...
UbikMobile. Figure 1 shows the basic components of the UbikMobile architecture for manual and automatic testing. An overview of its main components is presented below.

– **UbikEditor** is a tool based on Sweet Home 3D[^6] to model the world where the simulation takes place. Its output is a file (*world.ubiksim*) which includes an environment, devices and simulated users. This file is used as a configuration specification for testing the SUT on a simulator.

– **UbikMobileSimulator** is the simulator and is based on *UbikSim* [29]. This simulator, which is the core of the whole UbikMobile architecture, receives a simulated world file as input and allows simulated entities and real elements (such as a developer or a smartphone) to interact between them.

– A SUT is executed on an Android device after compiling it with the APIT library. APIT stands for Application Programming Interface for Testing.

– The APIT library replaces Android services such as Camera in order to trick the SUT into considering the simulated world as the reality. Therefore, this library is a layer under the SUT which offers services related to sensing environment context from the simulator.

– **Sensor Services** are a set of modules that offer services to enable a simulator (the architecture allows different simulators to be used although this paper only employs UbikMobileSimulator) to send readings of simulated sensors. These values are forwarded to the SUT through the APIT.

– **R-OSGi** [6], Remote OSGi, is a module that allows communication between modules installed on different virtual machines. By this module, UbikMobileSimulator is able to use Sensor Services of a smartphone. Note that R-OSGi is not a contribution of this paper.

Each simulated smartphone is equipped with a set of sensors. The data of such sensors is generated in *UbikMobileSimulator*, depending on the state of the simulation. Different data streams generated from the same simulated device are forwarded concurrently to the corresponding real device through the architecture. Thus, the SUT installed on the real device use data from virtual sensors as if they were coming from the real ones thanks to the APIT. For example, a SUT in the form of a location service based on information from a couple of sensors, can be tested by generating sensor data indirectly from a simulated scenario. Modelling this scenario by using the *UbikMobile* architecture is more intuitive than defining the scenario as two time series for the sensors involved.

### 4. Modelling a simulated world

This section presents *UbikEditor*, the module of the UbikMobile architecture which allows developers to model an environment which will be employed to

[^6]: Sweet Home 3D website: [http://www.sweethome3d.com](http://www.sweethome3d.com)
test Context-Aware Services based on smartphones. UbikEditor, see Figure 2, is focused on indoor environments such as houses, offices or hospitals.

The tool is composed of four panels: a top-left panel is a palette where entities to create the world are available; a top-right panel is a canvas where the entities of the world are set; a bottom-left panel keeps a list of entities inserted in the canvas; and a bottom-right panel shows a 3D view of the modelled world. Indoor spaces can be built in an easy and precise manner.

Each element added in the canvas has graphic and semantic properties. They can be modified by double-clicking at them. Semantic properties are the most interesting for this work because they allow UbikEditor to: (1) name an entity to identify it during a simulation, (2) set the role of a virtual human to change her simulated behaviour and (3) link a virtual smartphone to a real one. Besides, there is a metadata text field which can be employed as a parameter with a purpose specified by the developer. For example, the string: “weight=80” could be used to set the weight of a simulated human. Finally, UbikEditor allows new functionality to be added by plug-ins. For example, figure 2 shows a plug-in to generate QR Tags (see details in section 7) and add them to the palette.

Thus, UbikEditor offers an efficient tool to model simulated environments. Thanks to this simplicity, a ubiquitous computing scenario can be developed by clicking and dragging elements (e.g., a room can be created by clicking four times on the canvas). See UbikEditor presentation video on [1]. The next two sections cover the remaining components of the UbikMobile architecture.

5. Connecting an Android device into a simulated world

This section presents the par of the UbikMobile software devoted to enable testing SUTs running at smartphones, recreating the real world by artificial sensor readings which are used by the real smartphone as if they were generated by the real corresponding sensors. For this, the following requisites must be satisfied:

– the SUT’s source code should not be changed from testing mode to production mode and
– the UbikMobile architecture must be easily uninstalled from the smartphone once simulation based testing is done.

5.1. Connecting the simulator and the smartphone

A flexible and modular architecture is required to allow developers to handle the ever changing nature of functional requirements. To meet this flexibility, the OSGi advantages are many [5], such as a lightweight execution core valid for systems with limited resources like a phone, dynamic extensibility to suit different requirements, or service-oriented programming within a virtual machine (VM). OSGi is used in a number of systems to handle context-aware services reliably and securely [18]. In these systems, context information is easily discovered, acquired, and used for reasoning thanks to OSGi. Each module can be installed, stopped, started and uninstalled without rebooting. Besides, a module can publish services while other module can monitor and use these services when they are available. For example, thanks to OSGi, a sensor can publish context information given by its read values while a reasoning module receives changes in the state of the former to use them.

But these services published by a real device in order to receive simulated sensor data values are not accessible by a smartphone agent in a simulator because both run on different VMs. This problem is solved by using Remote OSGi (R-OSGi) [6] since it allows modules to use services that are running in other VM. R-OSGi is implemented as a module which manages the publication and use of remote services, such as offering Apache CXF Distributed OSGi subproject\(^7\). This subproject gives the Reference Implementation of the Distribution Provider component of the OSGi Remote Services Specification [6]. Besides, R-OSGi does not modify the way, from the code point of view, in which services are published and used. Therefore, R-OSGi has been chosen as mechanism for establishing communications between the UbikMobileSimulator and a real device through remote services.

\(^7\)Apache CXF website: http://cxf.apache.org/distributed osgi.html.
5.2. UbikMobile modules in the Android device

This section presents the design of modules which are in the real device and which are required to manage simulated sensors readings sent by the UbikMobilSimulator.

As shown in Figure 3, some of the modules presented in this section are built on OSGi (R-OSGi and Sensor Services), and others are not (APIT and SUT). Since Android applications cannot be built as OSGi modules following the standard methodology and using the Android OSGi framework would require changes in the SUT code, the UbikMobile architecture does not use OSGi for the SUT and APIT.

OSGi modules are focused on publishing services to receive simulated sensor data, manage them and send them to the APIT. APIT is just responsible for offering the same interface as Android SDK to a SUT and presenting simulated sensor data in an indistinguishable manner.

5.2.1. Sensor Services

Sensor Services are a set of OSGi modules including an interface module and its corresponding implementation for each sensor related service of the Android SDK. Each module is responsible for publishing and managing the state of its service. For example, Figure 3 illustrates how the module CameraImpl implements the interface Camera to receive images from the simulator. This module has to resize the received image to match it with screen resolution of the device. Besides, depending on what type of image has been required by the SUT, its format has to be changed. Therefore, it could be said that Sensor Services replace sensor drivers in the smartphone.

A special module of Sensor Services is BridgeService. Its responsibility is to connect OSGi modules with the SUT through intents. An intent in Android is an abstract description of an operation to be per-
formed. Communications between OSGi modules and Android modules, which have a different philosophy of interaction as explained above, are centralized in BridgeService in order to allow developers to modify this communication easily in new versions of both platforms.

This design, which separates interfaces from implementations, enables an implemented module to change dynamically even without stopping the system. For example, it would be possible to have multiple versions of a module with different manners of publishing their services (e.g., through R-OSGi or a Web Service). Therefore, another simulator, which is not built on OSGi, could reuse this part of the UbikMobile architecture by invoking Sensor Services in its own manner (e.g. by Web Services or TCP/UDP).

5.2.2. API for Testing: APIT

The APIT (API for Testing) is a library used by the SUT to interact with the BridgeService. This library is the key to get transparency from the source code perspective because it connects the SUT with the rest of the UbikMobile architecture. The SUT is built upon this layer, as shown in Figure 3. Both the SUT and the APIT are compiled together compounding an Android application which runs on Android OS.

A class diagram of the APIT is shown in Figure 4. Android applications are composed of one or more classes extending the Activity class. Each activity of the SUT extends APIT Activity class which, in turn, extends the original Activity class. The method getSystemService(SERVICE_TYPE) in this APIT class gives access to most of the SDK services related to perception. The method is redefined to return APIT or the Android SDK classes depending on the type of service required. For example, if SENSOR_SERVICE is passed as parameter, the method returns APIT SensorManager class. But if WIFI_SERVICE is passed as parameter the method returns WifiManager class, which belongs to Android SDK. Other services of SDK related to perception are obtained in a different way, without using Activity class. For example, the Camera class shown in Figure 4 illustrates that there is no link between this class and Activity class. An instance of the Camera class is obtained using its open() static method instead of the getSystemService() method. Thus, the result of this design is that all Android SDK classes related to the environment perception are replaced with APIT classes with the same name, the same methods (interface), and the same package name, except that the prefix sim is included. Those classes receive simulated sensors values through Sensor Services instead of drivers of the sensors of the device.

In summary, connecting a SUT in a smartphone with a simulated environment to conduct testing tasks entails: (1) having the OSGi mobile framework with all modules installed; (2) importing the APIT library; and, (3) modifying slightly some SUT classes to add the sim prefix in the imported packages. Therefore, the APIT introduced in the UbikMobile architecture meets the requirements specified in section 3, since only very slight changes are required in the SUT (only the prefix of some packages).

5.3. UbikMobileSimulator

UbikMobileSimulator extends the UbikSim simulator with several features to carry out the manual testing of context-aware services based on smartphones. Specifically, it includes:

- a mechanism to control avatars from an input device such as a mouse or a keyboard.
- a smartphone agent which represents the simulated device and sends simulated sensors values to its corresponding real smartphone. The smartphone agent is modelled as an object in the environment which implements a Portable interface allowing users agent to carry it or drop it. Each one of these smartphone agents has a set of sensors associated to perceive the simulated environment.
- a set of modules in order to simulate the sensor devices mentioned, such as a camera sensor. In some instances, what is perceived by simulated sensors is directly taken from the environment, such as the images from the camera or device orientation. In other cases, they must be generated, such as inertial sensors (accelerometer and gyroscope). These simulated sensors should generate realistic enough values to produce a trustworthy testing of the SUT.

The following section covers more modelling features of the UbikMobileSimulator to model an artificial society of users which enables the UbikMobile architecture to conduct automatic testing of SUTs in real smartphones.

The use of other input devices such as the Kinect is being considered.
6. Agents modelling to automatize tests

The previous sections detail how to simulate a realistic scenario and connect a smartphone for testing mobile applications into a simulated world. However, the developer must play the role of the moving user and, therefore, launching an automatic series of tests is not possible. For automatic testing, modelling phone users and their interactions (user-environment, user-user, and user-smartphone) is necessary. This section introduces the tools provided for this purpose in Ubik-MobileSimulator.

Agent Based Social Simulation (ABSS) has become one of the most popular technologies to model and study complex adaptive systems such as, in this paper, users of a Context-Aware service [29]. ABSS combines computer simulation and social science by using a simple version of the agent metaphor to specify single components and interactions among them. The basic methodology to study a system by ABSS involves [17]: (1) studying the target system, (2) modelling it, (3) implementing a simulation, and (4) studying the results after executing the simulation. Surprisingly enough, although the interaction among agents is the key of most of the systems studied by ABSS, the most popular frameworks to develop these simulations do not offer any mechanism to provide developers with tools to define these interactions. Examples of these frameworks are MASON [19], Repast [13], and NetLogo [2]. On the other hand, in the field of Multi-agent systems (MASs), the definition of abstract communications among agents by interaction protocols is usual. In this way, the Foundation for Intelligent Physical Agents (FIPA) is devoted to promoting the interoperation of heterogeneous agents and the services that they can represent. A number of MASs frameworks which allow FIPA interactions protocols to be defined in an easy manner have been expanded recently to be used for the development of ABSS. Some example of these are: Jade [22], INGENIAS [26] and Zeus [32]. The main disadvantage is that the agents implemented in these platforms usually need considerably more resources than their counterparts in ABSS frameworks.

Nevertheless, even when considering a FIPA compliant framework, the need for new tools to define agents’ behaviours is justified by the amount of decisions required to replace users of context-aware applications with agents. Although there are significant contributions for defining agents, since most of them are not devoted to emulating humans and their interactions, they usually lack already implemented solutions for this problem. For example, defining a conversation between agents following the FIPA request protocol is usually straightforward in most MASs platforms.

---

Fig. 4. Class diagram which illustrates libraries used depending if application is compiled for testing using a simulator or for deployment in a real environment.

---

9FIPA website: http://www.fipa.org/.
However, if, for example, the conversation takes place in the context of an agent representing a simulated user which interacts with a smartphone agent, there is a lot of extra design and implementation to be carried out in order to check if the user is carrying the device, check if the device is available to receive a message (it could be locked), check if the message is received (the user could try to order something by touching the device but it might not react), decide the frequency in which the user has this conversation to emulate a realistic use of a context application, etc. Our architecture not only aims to classify these recurrent decisions, but also to offer implementations for them which can be easily used, reused, and extended.

6.1. Modelling agents’ behaviours

UbikMobileSimulator provides developers with a module called hierarchical automaton of behaviours (HAB) [11] that allows users of smart-phones to be modelled. This same HAB can be employed to simulate the behaviours of electronic devices or any agent involved in the simulation.

In essence, a HAB is composed of: a list of pending transitions ordered by priority, a method for creating new transitions (adding them to the list), a current state (state with the control) and a default state (state that takes control of the automaton when the list of pending transitions is empty). An interpreter [15] makes the automaton advance every step of the simulation. The main actions of the interpreter are described below. (1) First, new pending transitions are generated from the current state to another. (2) Then, the interpreter checks whether the next state is completed (or still empty) to undertake the next transition in the list, the one with highest priority. (3) If the list is empty, the default state is taken as current state. (4) If the current state is not finished but there is a transition with highest priority in the list, the interpreter gives the control to the latter. Finally, (5) the current state takes control to undertake an action. In our proposal, the HAB control is performed transparently to the developer. The source code of this interpreter can be consulted on-line [1].

Hierarchical automaton means that each state is itself another automaton, a subordinate automaton, and therefore, it has the same components. By default, the modules presented assume that an automaton ends and returns the control when the time assigned as a parameter in its creation is over, or if there is no default state given and no transitions pending. Note that the main automaton (level 0), the highest in the hierarchy, never ends if a cyclic behaviour wants to be modelled (and therefore a default state must be given). UbikMobileSimulator offers the class Automaton to define a HAB extending it. The tasks to be implemented by the developer are: (1) creating new transitions, (2) getting the default state, and (3) including an ending condition for the automaton if needed.

The states at the bottom of the hierarchy (with no subordinate automata) implement simple behaviours. The developer must create one of these simple states when defining an action which will be performed in one step to immediately return control to the automaton father. The class that defines simple states in UbikMobileSimulator is SimpleState. (1), (2), and (3), methods explained in the above paragraph and needed in the definition of Automaton instances, are not necessary for simple states because they do not have: transitions, a default state, or a final condition. These states require redefining only one method step which carries out an “atomic” action (in the sense that modelling an automaton is not necessary due to its simplicity).

6.1.1. Graphical modelling of an agent’s behaviour

This section introduces a graphical notation for modelling behaviours among agents that can be immediately translated into classes provided by UbikMobileSimulator. The notation does not aim to show all the details of the final implementation but to include the major design decisions. As with the use of UML for general software systems, there are a number of reasons for undertaking a model of agents’ behaviours. First, modelling a part of real world is always complex and, since programming is very time-consuming, a visual model to be approved before coding is useful. Second, a visual notation allows agents models, which are very abstract and hard to visualize, to be visualised...
in multiple dimension. Third, this notation accommodates incremental development and re-development, since the visualization of the different modules of the model allows developers to extend, reuse, and maintain it in an easier manner. Finally, although the notation does not produce code automatically, which will be described in future works, we are working on this automatic code generation and on an automatic verification from a visual editor.

The notation introduced in this section defines a graph with the main agents involved in the simulation and their behaviour. The types of node in this graph are listed below.

- **Simulation.** Only one node for each scenario; it is the root of the graph.
- **Agent.** Nodes connected to simulation. These are agents that have an Automaton governing their behaviour.
- **Roles.** Nodes connected to agents. These nodes are implemented internally as an Automaton and allow the same agent to have different behaviours for different situations (one person can have a number of behaviours when playing the role of worker and another when the role of client is being played).
- **Behaviours.** Nodes that have incident edges from other behaviours or roles, indicating that the former are subordinate automata of the latter, and can have outgoing edges to other behaviours, indicating that the latter are subordinate behaviours of the former. When a behaviour node does not have outgoing edges, this node can be a SimpleState, with an atomic action to be implemented by developers, or a previously defined and implemented Automaton.
- **Interacting behaviours (IB).** Special kind of behaviour node, shown in grey, which involves interactions among several entities of the system under test. This kind of behaviour will be detailed in section 6.2.1.

Edges in the graph express a transfer of control from one node to another (an Automaton delegates in a subordinate Automaton to implement a behaviour). Several outgoing edges from a node involve evaluating the possibility of generating transitions to the destination nodes. Edges are labelled with a tuple \(x, y, z\) indicating: (1) the order used to evaluate the possible delegation of control, \(x\); (2) the condition to give the control, \(y\); (3) and the number of times the source node can give control to the destination node, \(z\) (\(z = n\) to not limit this number). \(x\) and \(z\) are natural numbers. The transition function \(y\) is local to each edge and can take the following values:

- **Prob**. The transition is taken with a probability of \(p (0 \leq p \leq 1)\) in each step of the simulation.
- **Nonmon**. The transition is taken or not according to a probability distribution for non-monotonous behaviours. These are the kinds of behaviour the subject usually manifests, when not bounded by a concrete time slot, and repeated within a non-constant period (e.g. going to the toilet, having a shower, cleaning the house and so on) [15]. In previous work, and after examining empirical data of actual people studied, we concluded that non monotonous behaviours, such as going to the restroom, fit an exponential probability distribution function [15]. Note that, as explained above, behaviours are related to nodes while edges specify when a transition is produced from one node to another (with these nodes being: roles, behaviours or interacting behaviours). Note also that nonmom, as an edge parameter, is defined locally for the edge with specific parameters for the transition being modelled.
- **Mon**. The transition is taken or not according to a probability distribution for monotonous behaviours. These are the kind of behaviour the subject always manifest approximately in the same time slot, and on a daily basis (e.g. sleeping, having meals, medication and so on). Previous work shows that these behaviours, such as having lunch or sleeping, fit a gamma probability distribution function [15]. UbikMobileSimulator offers the class “PDF” (Probability distribution function) which implements constructors and methods to launch or sleeping, fit a gamma probability distribution function [15]. UbikMobileSimulator offers the class “PDF” (Probability distribution function) which implements constructors and methods to launch or not monotonous and non monotonous behaviours at appropriate times.
- **Notsp**. Not specified, the condition to take this transition is different from the previous ones. The condition has to be implemented in the Automaton instance of the source node of the edge.
- **Def**. Default state, transition taken only if the rest of the possible transitions from the current node are not allowed. Therefore, the \(x\) value, which as explained above is the order used to evaluate the functions in different outgoing edges of a node and to decide which transition happens, is ignored when \(y = Def\). Note that this same \(x\) value is used to select the transition to be undertaken
when several of them are possible according to the \( y \) function (nonmom, mon, etc.).

There are default values for the three fields of an edge label \((x, y, z)\) which allows them to be omitted. \( x \) not specified indicates that the evaluation of the different outgoing edges is undertaken in a random order. \( y \) by default indicates a probability of 1 (the transition is always generated). Finally, \( z \) unspecified means that, for the current execution of the source node, the control only can be given to the destination node once \((z = 1)\). The edges between Simulation and Agents are not labelled at all since these elements are not implemented by the Automaton class.

Figure 6 graph shows an example of behaviour for children which could be used, for example, to test context-aware systems for enhancing child development [34]. In this example, the simulation consists of two agents, whose names are Joseph and Alicia, who can play the role of children (unique role played by default), and which consists of two behaviours: GoToBedAndSleep and GoAround. The first of these is a monotonous behaviour while the second is a default behaviour that is taken when the child is not sleeping. GoAround consists of three behaviours of movement to go to room, living room and yard respectively. The order of evaluation to generate these three possible transitions is sequential \((x\) marks the order). Besides, the evaluation is carried out using an exponential probability function \((y\) for non monotonous behaviour). Finally, no limit on the number of transitions generated to each sub-behaviours is given \((z = n)\) for the two first behaviours, and a limit of 2 is given for “MoveToYard”.

This graphical model describes a top-down approach which involves the breaking down of a behaviour to gain insight into its compositional sub-behaviours. Code reuse is a major concern of the modules presented. Before the basic classes to be extended for modelling behaviours (Automaton and SimpleState), some example automata implemented in UbikMobileSimulator that can be used as defined behaviours to be combined and to create new behaviours are: SimpleMove (to allow an agent to move in a straight line); DoNothing (do nothing, typically used as default state); Move (move to a room or a particular position establishing automatically an optimal path in the map), this automaton has several subordinate SimpleMove automata; MoveAndStay (which allows an agent to move to a room and stay in a certain time), this automaton has as subordinate automata DoNothing and Move.

6.2. Modelling agents’ interactions

This section details the hierarchical automaton of interactions (HAI) [11]. This module of UbikMobileSimulator is combined with the HAB discussed above in order to model social interactions. The HAI is a hierarchical automaton in three levels: InteractionsHandler, Conversations and Protocols. Besides, Protocols have three different levels of abstraction, see Figure 5.

The control of the HAI is illustrated in Figure 5. An initiator, an agent which wants to start an interaction, calls the InteractionsHandler to register a conversation with a protocol and a channel. The handler passes control to the conversation in every step of the simulation. The conversation passes control to a protocol (if it is allowed by the channel). The protocol produces messages in the conversation, marks the end of the conversation, and makes the participants react (usually causing new transitions in the HAB). The following section discusses these elements in more detail and explains the decisions required when implementing a HAI.

The main components of the HAI are: messages, protocols (three levels), channels, conversations and InteractionHandler. These are explained below.

The communication implemented is based on a Message passing, but this mechanism is used to model various types of conversations (voice, email, phone, etc.). Messages have the typical elements of a FIPA ACL message [3].
The Protocol is the message producer and has three different levels of abstraction. A first level is responsible for performing tasks that are implemented for a protocol regardless of the semantics exchanged. Its objectives are:

1. Providing the set of performatives, i.e. communicative acts [3], that can follow a given performative in the protocol.
2. Determining if the protocol is finished for a given message and if this end has been successful or not.
3. Verifying, given a conversation that has followed this protocol, if the protocol has been followed correctly.

A second level of abstraction is employed to define the specific purpose of the protocol. At this level, the problem is no longer deciding the next performative, but deciding the semantics or content for the following message. That is:

4. The semantics of a received message must be studied in addition to its performative to select the next message to be sent.
5. This level is also responsible for selecting a set of participants for the conversation.
6. Besides, this level reacts once the protocol has been completed, i.e., it changes the behaviour of the participants based on the results of a conversation.

Finally, at a third level of abstraction:

7. Each agent can have different preferences to fulfil a protocol, so there are as many instances of the third level as preferences needed.

The protocol produces messages assuming that the agents are in an appropriate state of the HAB and that the messages are correctly received. However, the messages may not always reach their destination. The requisites to decide whether a message is received or not are implemented in the Channel. In particular, the channel is responsible for five different tasks listed below.

1. Initialize participants reserves the use of the channel when necessary (e.g. for a phone call).
2. Channel free to send decides if the person modelled is available to send a message through the channel in a conversation (e.g. having a cell phone and not already calling).
3. Channel free to receive is also necessary because in some channels the proper sending does not imply that the recipient receives the message (e.g. modelling an interaction by e-mail).
4. Discard message when a message does not reach its destination at once (e.g. voice is ruled out, but an e-mail is not).
5. Finish participants undertakes tasks after the end of a conversation on the channel (e.g. releasing a channel reservation if it was made in (1)).

A Conversation uses a channel, a protocol of level 1 and 2, and, if necessary, specific protocols of the participants (level 3) to generate messages. The conversation is a protocol and “channels” interpreter. The conversations are registered by the HAB in the InteractionsHandler, another interpreter which manages the different conversations. These interpreters, whose source code is available on-line [1], act in a transparent manner without requiring additional code as the interpreter of the HAB explained in section 6.1.

6.2.1. Graphical modelling of agents’ interactions

Section 6.1.1 explains a graphical notation for modelling behaviours in UbikMobileSimulator. This section focuses on the interacting behaviours (IBs) which involve interactions among agents. The IBs include a series of special nodes and edges that offer information about the type of interaction. These labels and the nodes that connect are specified below:

- **Part.** Participants of the conversation besides the initiator. The node connected can be an agent or a role. Several edges can be included to indicate several participants.
- **Prot.** Protocol executed. The node connected is a protocol of level 1, indicating the sequence of messages.
- **Chan.** Channel of the interaction. The destination node has to be an instance of Channel.
- **RI.** Reaction of the initiator, this edge connects the IB with another behaviour that the initiator performs if the interaction ends successfully according to the protocol specification. This reaction is specified in the level 2 protocol but agents can specify their own reaction in the level 3.
- **RP.** Reaction of participants, this edge connects the IB with another behaviour that the participants undertake if the interaction ends successfully. Again, this reaction is specified in the level 2 protocol but participants can specify their own reaction in the level 3.
6.3. Providing smartphone agent to smartphone interactions

Section 5 introduced a test approach where the developer played the role of the user for the context-aware service under test. Nevertheless, to conduct automatic testing by using users modelled by agents, these agents have to be able to interact with the smartphone. This section explores the modules needed in UbikMobileSimulator to make this possible.

There are two fundamental requisites for carrying out automatic testing: (1) Agents must be able to act on the input methods of a device, such as physical buttons or tactile screen and (2) they must be able to understand the outputs of the device, such as elements displayed on the screen. A simulated smartphone is modelled as a smartphone agent, see Figure 8. Thus, a communication between a user agent and a smartphone is in fact a communication between two agents. The interaction of a smartphone agent with its real device is possibly due to Chimpchat library. It offers an API that allows developers to: write scripts for launching applications; generate events related to pushing physical buttons or touching a position on the screen; or take a screenshot of the screen. Although this library is not a contribution of this paper, its inclusion as a module of UbikMobileSimulator is necessary to provide the architecture with smartphone agent to smartphone interactions.

Figure 8 shows the use of Chimpchat to connect the smartphone agent to a smartphone. Chimpchat is able to manage several emulators and devices connected by Universal Serial Bus (USB) interface at the same machine. Therefore, the number of real devices used is limited by the number of USB interfaces of the machine, but not the number of emulators. User agents interact with smartphone agents using the HAB and HAI modules presented in sections 6.1 and 6.2, respectively. Each smartphone agent is associated with only one real smartphone. These agents use an Interface for Interacting with Android Devices (IIAD). This layer has a higher abstraction level than chimpchat, offering more elaborated functionalities such as unlock a device. This task is translated into functions of chimpchat, in this case into a drag(starting point, end point function which simulates a drag gesture (touch, hold, and move) on the device’s screen. Note that not all devices are unlocked in a similar manner or have the same screen size. Therefore, different IIAD layers may be required by mobiles agents to be employed for different smartphones.
7. Case studies

This section presents some case studies to illustrate the use and possibilities of the UbikMobile architecture presented in this paper. Case studies are related to an augmented reality (AR) application for a museum located in the city of Murcia and called Museo Arqueologico de Murcia (MAM). This application is conceived to make museum tours more attractive and educational by locating POIs (Point Of Interest). AR applications need to know the location and orientation of the phone in order to show the POIs on the screen. Location is achieved by reading QR tags which contain its location coded. Therefore, a camera sensor is necessary on the device. Orientation is achieved using accelerometer and digital compass sensor available in the smartphone.

Note that, as in the example presented in the introduction, there is a correlation between the tested service and the data readings from various sensors in this case study. POIs and images displayed on the screen depend on location and orientation of the device and its camera. Hence, the SUT requires correlated data of three sensors (camera, accelerometer, and digital compass)\(^{10}\).

This section illustrates different functionalities of the UbikMobile architecture. First, (1) UbikEditor is employed to create a basic model of the exhibition hall of the museum. Second, (2) UbikMobile is used to allow a developer to control a simulated avatar carrying a real smartphone. The smartphone is tricked into thinking that the simulated world is the reality, so allowing a manual test to be undertaken. Third, (3) a user model is created and implemented to perform automatic testing. This model includes user-environment, user-smartphone and user-user interactions. Finally, (4) the realism of our architecture for a QR reader case is tested.

7.1. Modelling a simulated museum with UbikEditor

Figure 9 shows the editor and a model of an exhibition hall of the MAM. Each world’s model represents a test configuration. Hence, first a basic environment is created with exhibition hall, furniture and pieces of art. As shown in Figure 9, the base scenario for testing was modelled using a plan of the museum as a basis. Thus, walls, floor of the room and showcases were traced quickly. Then, the room was decorated by importing 3D objects. This basic model can be employed in different tests. Elements for a specific test can then be included, such as QR tags with different sizes to check if the required distance to read these codes is appropriate. For this purpose, some QR tags were added using the plug-in QR described in section 4. Next, case studies use this simulated environment.

7.2. Manual testing of an AR application

This section illustrates the manual testing of the AR application for the MAM presented above. For that purpose, some QR tags coding their location in the museum were added to the scenario using UbikEdi-
tor. Secondly, an AR application was developed using a LookAR Framework\textsuperscript{11}—this is the SUT. Giving a location and an orientation of the device to the AR application makes it possible to show POIs on the screen. Location is obtained by reading QR tags using the software Barcode Scanner\textsuperscript{12}. This scanner is a second SUT. Thus, two simulated sensors necessary: camera and orientation. The AR application uses these two sensors and Barcode Scanner uses just the camera sensor. The camera sensor allows the simulated environment on the screen to be seen and QR tags to be read. The orientation sensor offers the orientation of the simulated device (azimuth, pitch and roll). Both SUTs were modified by adding the \textit{sim} prefix in the necessary import sentences to be compiled using APIT, see section 5.2.2. Finally, the manual testing of both SUTs was performed. The functionalities tested were: (1) QR tags are readable, with a given size, (2) POIs are properly located on the screen, (3) information attached to each POI is correct. A simple execution of the simulation allowed us to check that the QR tags size was correct for an appropriate reading and that the POIs were properly located on the smartphone screen. However, this first simulation detected faults in the matching between POI and its information. These faults were corrected and tested in a second execution of the simulation. This testing is manual in the sense that the developer plays the role of the context-aware service user and that, therefore, the developer has to witness the experiments and check manually if the visual information given by the smartphone is correct. Figure 10 shows a capture of a video available on-line\textsuperscript{[1]} which displays the testing process provided by the UbikMobile architecture in this case.

7.3. Modelling a user model interacting with a real smartphone

This section illustrates the modelling of users to test automatically the two SUTs tested manually in section 7.2. For this purpose, the HAB and HAI (modules of UbikMobile) are programmed according to the model shown in Figure 11 which follows the notation introduced in sections 6.1.1 and 6.2.1. This model describes an agent called Bob which takes “visitor” as default role. This role involves a sequential behaviour: move close to a new QR code, read the QR code, spin with the smartphone to detect POIs, get a list of POIs and visit the different POIs. Note that default values are used for setting the number of times the node can pass control to the destination node and to specify the condition when control is passed. Therefore, a single transition to the destination node is always generated. “GetPOIs” is an interacting behaviour where the participants are the agent and the smartphone agent (connected to a smartphone as described in section 6.3), the channel is “view”, the reaction of the smartphone agent is to show a list of POIs, and the reaction of the agent (initiator) is to store that list to be used in the next behaviour. If a behaviour does not have a trivial implementation by using the predefined behaviours given by UbikMobile, following the top-down design, more details are given. Thus, “visitPOI” is composed of: go to a POI, push the icon of the POI in the smartphone, push the icon for getting more information about the POI, push back to allow the smartphone to detect new POIs, and observing the POI whose information has been given by the smartphone. Note that visiting a POI is also a sequence behaviour ($x$ parameter) where $y$ and $z$ take their default values and where several interacting behaviours are involved. These users model can be employed to test a number of features of the SUTs automatically: are all the POIs visible from the QR codes positions?, are different QR codes redirecting to the same POIs?, are users hindering other visitors from detecting or visiting the POIs?, is the information given by the service correct for each POI?. These questions are easily answered by keeping a list of POIs which have been visited for every user after reading each QR code. A simple execution of the sim-

\begin{itemize}
\item[11] LookAR website: \url{http://www.lookar.net/}.
\item[12] Zxing and Barcode Scanner website: \url{http://code.google.com/p/zxing/}.
\end{itemize}
Fig. 11. Graphical modelling for a museum visitor interacting with a smartphone.

ulation with one modelled user allowed us to prove that all the POIs were visited and, hence, they were all visible from the QR positions (the list for the user contained all POIs). On the other hand, different QR codes redirected to POIs already visited by the user (there were repeated elements in the aforementioned list). Finally, we repeated these two tests but including ten users in the simulation. The results showed that all QR codes were visited. However, for some users there were not repeated elements in the list of visited POIs. The conclusion is that this layout of QR codes and POIs includes redundancy for only one user (same POIs can be detected from different QR codes). Nevertheless, this redundancy is useful when more users are considered because some POIs will not be detected by a user as a result of the presence of other users between the smartphone and the POI. A video available on-line [1] shows an execution of a simulation with this user model which is capable of combining simulated users with real smartphones.

7.4. Validating the simulated camera sensor

This section tests the module responsible for reading a simulated QR code using a real smartphone. The experiment entails finding the maximum distance that allows the smartphone to read QR codes with different sizes and angles. This experiment was conducted in both the real world and UbikMobile in order to compare both results and to validate the realism of our architecture for this case.

The QR code reading application is the open source tool Barcode Scanner, used in previous case studies.
The experiment involves capturing the maximum distance to read a QR code varying the angle of incidence with the following values: -45°, -22.5°, 0°, 22.5, 45.5°. Experiments were also conducted for different sizes of the QR labels: 1cm, 2.5cm 5cm. In the simulated scenario, a smartphone (Samsung Nexus S) operating system was compiled with the API of the UbikMobile architecture, three different scenarios were built with UbikEditor, and each scenario included the same QR code with a different size. In the real scenario, see Figure 12, a QR code with the same content as in the simulated scenario was employed and measurements were taken three times for each size and angle considered.

Figure 13 shows the results for both scenarios. The x-axis represents the angle of incidence and the y-axis the maximum distance (in cm) at which the code was read. A number of conclusions are extrapolated from this experiment. (1) The bigger the QR label is, the shorter the maximum distance at which the code can be read if the angle of incidence grows. This is due to the change of perspective and explains the bell shape of the bars in the chart. (2) QR codes are more legible if they are read from the left than from the right. This fact has been discovered because, for the same angle of incidence (e.g., 45°), the application was able to read the code from the left but not from the right. (3) Finally, the similar results obtained for both scenarios validate the realism of using the UbikMobile architecture to test applications where the camera is involved in general and for reading QR codes in particular.

8. Related works

A number of approaches dealing with SUTs in simulated environment can be found in the literature. A well-known tool is Ubiwise [8]. Ubiwise focuses on the use and analysis of environment models for ubiquitous computing systems. For this purpose, sensors and its communications can be defined. People are considered individually on a simulation engine based on Quake II, where real people, acting as players in a game, generate information about their own context, which is captured through simulated sensors. Multiple users can link up to the same server to create interactive ubiquitous computing scenarios. However, the virtual subjects are not autonomous, since they must be controlled by the users. The subjects interact with the environment in order to validate its deployed services.

TATUS [25] is another tool which allows experimentation of adaptive ubiquitous computing systems. It is based on a graphic engine called Half-life. In this case, the main novelty is that multiple SUT may be connected to the engine. The SUT is adaptive so it makes decisions about changing its behaviour in reaction to user’s movements and behaviour. Other environmental factors such as network conditions, ambient noise or social setting can also be considered.

In this case, some virtual subjects could behave autonomously due to simple AI scripts, although the tool is mainly focused on user-controlled characters and their interactions with the SUT.

Another interesting tool is UbiREAL [23]. It lets users intuitively grasp how devices are controlled, depending on temporal variation of contexts in a virtual space. The main contribution of UbiREAL is that it simulates physical quantities (e.g. temperature or humidity) and includes a network simulator. This network simulator allows communication between virtual devices and real devices. This is a very important aspect since it is possible to inject reality into the simulations. The behaviour of the virtual subjects must be preconfigured (it is possible to define a route and some actions to perform).

Tang et al. [31] propose a user-centred design methodology that combines the notion of situation (a description of the states of relevant entities) with an application model. Additionally, they offer a domain-specific design language and a set of graphical toolkits covering the life-cycle development of a pervasive application to support the methodology. OpenSim is the simulator employed and each entity is represented as a plug-in module. Users in the simulator are avatars and, therefore, they must be controlled by humans. Nevertheless, by recording context traces, experiments can be repeated without human intervention. A middleware is used to give support for executing the application, including the discovery of available services.

Roalter et al. [28] present a tool chain to simplify research on Intelligent Environments. The tool chain consists of: a middleware (Robot Operating System tool or ROS), environment edition (SweetHome 3D tool), and simulation and visualization (Gazebo tool). The ROS middleware plays an important role in making interactions possible. It offers: (1) basic communications methods; (2) a large community improving device drivers; and (3) compatibility with a large pool of hardware systems. Simulations of the environments

\[13\text{Open Simulator website: https://www.opensimulator.org.} \]
Table 1 presents a summary of some relevant characteristics for testing context-aware applications through simulations. These characteristics are: (1) allowing a real smartphone and/or an emulator to be connected with the simulator in a transparent way; (2) including a user model which allows developers to define user behaviours in the simulation; (3) injecting real elements, i.e. real elements such as sensors and real software can be mixed with simulated ones; (4) including an application model to define the application behaviour in an abstract way; (5) allowing applications to be developed by graphical programming; (6) being available online; and, finally, (7) being open source.

The approaches presented in this comparison are very significant contributions to the testing of context-aware services and they have been very useful for this work. However, several of these proposals have not their implementations or source code available online to allow researchers to extend and improve them.

UbikMobile, the architecture presented in this paper, is, to the best of our knowledge, the only simulation approach which allows developers: (1) to design a realistic simulated environment; (2) to connect real smartphones to the simulated environment in order to perform testing tasks; and (3) to model the users of the context-aware applications under test giving them the capacity to interact with the smartphone.

9. Conclusions and future work

This paper presents the UbikMobile architecture which aims at allowing developers to test context-aware services based on smartphones in a simulated environment. This approach is motivated by the cost and difficulties of testing these services directly in Living Labs and a number of shortcomings in simulation approaches which, basically, lack usability and realism.

This paper has shown that UbikMobile and its different components enable developers to:

- Design a simulated environment graphically in an intuitive manner by UbikEditor.
- Connect a real smartphone (or any device with an Android OS) into the simulated environment. This feature is achieved by the APIIT and Sensor Services modules of UbikMobile which allow the smartphone to receive data from the simulation as if received by the sensors. This use of a real smartphone is the main contribution of the architecture since it enables a system under test (SUT) to be tested in its final hardware but without needing the final environment. Moreover, very slight changes are required in the SUT to be connected in the architecture (only names of packages imported). Therefore, the approach minimizes the
the possibility of including new faults due to this connection.

– Conduct manual testing were the developer controls an avatar in the simulation and the smartphone to interact with the SUT. UbikMobileSimulator is the component which, in combination with the elements previously mentioned, allows this manual testing to be performed.

– Conduct automatic testing were the role of the user (or users) is played by an agent based social simulation (ABSS). UbikMobileSimulator includes several tools to model users as agents. This approach, unlike most frameworks for defining agents and their interactions, offers a large number of implemented solutions to common problems in this emulation of users with agents. Two formalisms are introduced to model such behaviours: HAB and HAI. Besides, since simulated agents have to deal with real smartphones in this approach, tools such as Chimpchat have been employed to make this possible.

Reusability and extensibility, which are important concerns in UbikMobile, are addressed by employing OSGi and R-OSGi to allow different modules to communicate in a transparent way and to be replaced easily. Different case studies have shown the usability of the architecture.

Future works include using this architecture in a wide range of context-aware services and comparing its results to using Living Labs directly. We are also experimenting with Eclipse Modelling Framework (EMF)\(^1\) to translate the notation introduced to model agents’ behaviours and its interactions into code automatically. Some preliminary results are shown in a video on-line [1]. Finally, since the definition of agents can generate extra complexity and extra bugs, techniques of model checking are being studied to provide developers with an automatic verification of agents’ behaviours before automatically generating the code of the simulation with EMF.

Acknowledgments

This research work is supported by the Spanish Ministry of Science and Innovation in the scope of the Research Projects TSI-020302-2010-129, TIN2011-28335-C02-02, CEN 2010-1019, and through the Fundación Séneca within the Program 04552/GERM/06.

References


\(^1\)EMF website: www.eclipse.org/emf/.


