Device Event Based Test Automation Module for Smartphone

Jae-Ho Lee, Seok-Jin Yoon, Do-Hyung Kim
Mobile S/W Platform Team
Electronics and Telecommunications Research Institute
Daejeon, Korea
{bigleap,kimdh,sjyoon}@etri.re.kr

Cheol-Hoon Lee
Dept. of Computer Engineering
Chungnam National University
Deajeon, Korea
clee@cnu.ac.kr

Abstract—Many mobile companies have adopted Linux to their products. The Linux-based mobile platform is a very complex software stack consisting of three layers: the kernel, the middleware, and the application. The complexity of Linux-based software makes integration testing more difficult. Legacy testing has focused on testing APIs and GUI-based applications by manual input. There is no automatic way to achieve the integration test because of applications driven by sensors such as GPS, accelerometer, and so on. This paper defines the event types generated from each layer in a Linux-based mobile software stack, and proposes an event-based test automation system which is able to capture and playback events generated from hardware sensors as well as user input. The proposed system handles all events with a uniform interface at the kernel device level, which enables developers to achieve an easy and efficient integration testing in some automatic way.

Mobile Platform; integration testing; test automation; device event

I. INTRODUCTION

Over the past five years, mobile industry leaders have adopted Linux-based mobile platforms for their products such as smartphones, webpads, and tablets since the first Linux-based smartphone, A760 by Motolora, was sucessful in China. The mobile platform, based on a general purpose operating system (GPOS) like Linux, consists of very complex software modules which make integration testing difficult and time-consuming. Vendors need an easy and quick method of testing the entire software which includes the kernel, middleware and applications in order to shorten time-to-market. Current methods require the testing of individual units of source code to determine if they are correct for use and verifying GUI-based applications by a tester's manual input[1][2]. An off-the-shelf tool like TestQuest provides the ability for capturing and replaying the events from and to GUI components and uses a script-based scenario file, in order to support test automation.

Most software vendors distribute a Software Development Kit (SDK) to conveniently develop mobile applications running on their mobile platform. The Linux-based mobile platforms for the smartphone, Android, LiMo, OpenMoko also provide their own SDK which includes a phone emulator based on hardware emulation technology, which enables developers to test their applications without physical hardware(hereafter referred to as actual hardware)[6][7][8]. These emulators use QEMU which is an open source project and allow an ARM-based instruction set to run an unmodified mobile software stack on a guest operating system[5]. Fig.1 shows the concept of a switchable full software stack. A simulator based on hardware emulation separates the emulated mobile software stack from a desktop. This architecture allows the mobile software stack to be entirely replaced with another mobile platform and provides a virtual execution environment on a hardware emulation layer. The mobile software stack includes a target agent which interworks with a SDK plug-in for the test automation.

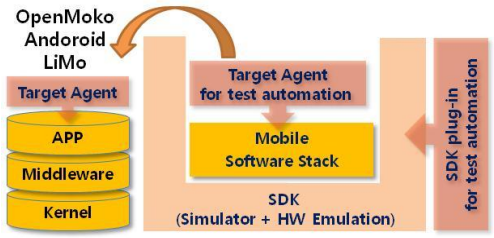

Figure 1. The architecture of a switchable mobile software stack

II. TESTING TOOL OF MOBILE PLATFORM

A. System Overview

Fig. 2 shows the concept of a test automation tool which provides the ability of logging and replaying the events generated from each layer in the mobile software stack, including kernel layer, middleware layer and application layer. We classify the events generated from each software layer into three categories: input event, IPC event and platform event.

- Input event has two types: user input and sensor input. User input is generated in the foreground running application by user input devices such as a keypad, a touch screen, a volume button and so on. Sensor input is spontaneously generated from hardware modules such as a GPS, accelerometer, received signal strength indication (RSSI), battery, etc.

- IPC event is communication messaging between system-level processes and user-level processes. We use D-BUS which is an inter-process communication (IPC) mechanism. Communication happens through a central server application called message bus system. The implemented test automation tool is able to collect predefined IPC messages through D-BUS [5].
- Platform event is dependent on an application manager in mobile platform. When the running state of an application is changed, the application manager notifies the lifecycle events such as installing, removing, upgrading, starting, stopping, pausing, restarting, termination and so on.

An emulator's software stack is exactly the same as an actual hardware's ARM-based binary image. It means the mobile software stack image for an ARM processor can be used for the emulator and the actual hardware.

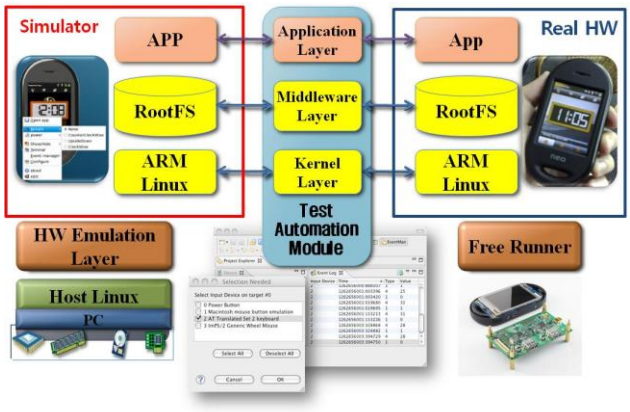

Figure 2. The concept of event-based testing tool

B. Test Automation Module

Fig. 3 shows the internal architecture of a test automation module which enables the capturing and playing of events generated from a mobile software stack. It largely consists of two parts: the Target Agent and the Test Automation Module (a SDK plug-in for test automation), as already indicated in Fig. 1. Target Agent is an application program to be installed on the mobile software stack. The Test Automation Module, which is made up of several plug-ins for eclipse-based SDK, is installed on the host desktop.

- Target Agent communicates with an external testing tool for logging and feeding a variety of events. In the logging procedure, events generated from the mobile platform are collected and sent to the test automation module. In the feeding procedure, previously logged events are received from the test automation module and injected into the designated software module.
- Connector sets up the network connection with an emulator or an actual hardware, depending on the configuration of the Test Execution Manager.
- Event Synchronizer controls the time interval during the event injection. The time difference of processing events is due to the gap of the computing power between an actual hardware and an emulator on the host. Synchronization is needed when the events logged from an actual hardware are injected to the emulator, or vice versa.
- Event Logger collects the events from the mobile software stack and saves them in a database (DB).
- Event Feeder injects the events into the mobile software stack to be tested through Connector and Target Agents.
- Scenario Manager creates a set of new test scenarios with the logged events and defines the combination of scenarios with previously created scenarios in DB. Test scenario consists of pairs (an event, the predicted outcome) which are used for comparing the actual outcomes of the injected events.
- Test Execution Manager converts events in the test scenario file into a transferrable format in Event Feeder and Connector. It also provides the functionality of managing a testing lifecycle such as starting, stopping, pausing, restarting, repeating and so on.

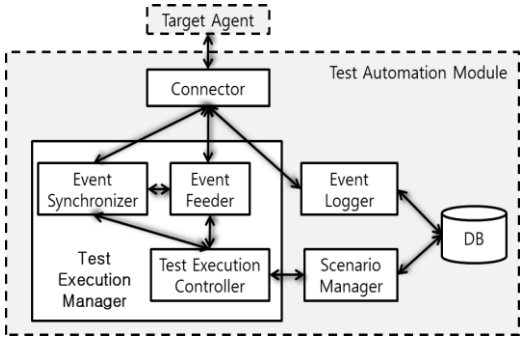

Figure 3. The Architecture of test automation module

At the beginning of testing, manual work is needed for creating and editing the test scenario, but once created and saved in DB, complex testing can be done with less human labor.

III. Test Automation based on Device Event

A. Event Device

User input devices such as a mouse, a keyboard and a touchpad along with sensor-based input devices such as accelerometer and GPS are registered in the Linux kernel and can be accessed with the generic input event interface through device files under "/dev/input." The Linux kernel provides a data structure for the input event as shown below[5].

```
struct input_event {
    struct timeval time; /*the Time at which the event happened*/
    __u16 type; /*event type*/
    __u16 code;/*event code*/
    __s32 value;/*the value the event carries*/
};
```

TABLE I. shows the events sequentially generated in the kernel while pressing a mouse button. This input event

mechanism passes the events with a timestamp generated in the kernel straight to the user program.

TABLE I. TOUCH EVENT IN LINUX INPUT DEVICE MODEL

No	Type	Code	Value	Meaning
1	EV_KEY	BTN_TOUCH	1	pressed state
2	EV_ABS	ABS_X	342	X-value
3	EV_ABS	ABS_Y	128	Y-value
4	EV_SYN	0	0	data end

TABLE II. shows an example of mapping physical devices into event device files in the LINUX kernel. This configuration is determined in runtime depending on the types of input devices an embedded system has. The proposed test automation module is able to manage sensor input data as well as user input data, by using the identical interface, which enables GUI-based applications and sensor data-driven applications to be tested one time. A sensor like an accelerometer generates too many input events, even by simple movement, which creates overload for testing nomal applications without using a sensor. In order to avoid this problem, the proposed test automation module provides the functionality of selecting devices to be monitored.

TABLE II. AN EXAMPLE OF EVENT DEVICE CONFIGURATION IN LINUX

Event Device File	Mapped physical Device	Event Source
/dev/input/event0	Keypad	User Input
/dev/input/event1	Touchpad	User Input
/dev/input/event2	Power Button	User Input
/dev/input/event3	Accelerometer(Sensor1)	Hardware Sensor
/dev/input/event4	Accelerometer(Sensor2)	Hardware Sensor
. . .	. . .	. . .
/dev/input/eventX	. . .	. . .

B. Eclipse-based Integrated Testing Environment

The proposed test automation module consists of eclipsed-based plug-ins and is able to connect both the hardware and a simulator at the same time, as shown Fig. 4.

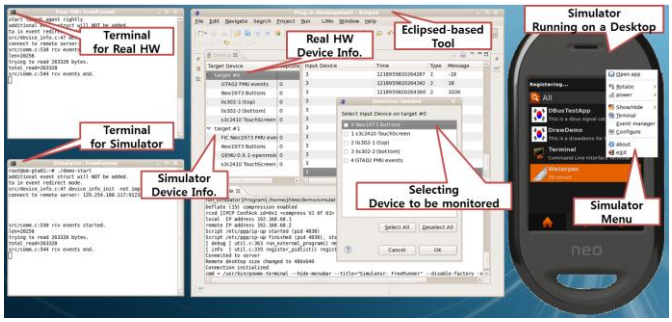

Figure 4. Eclipse-based integration-testing environment

Interworkings between the actual hardware and simulator provide developers with an efficient mechanism for test automation of their software modules, via several paths for event transmission. Event Logger records the events generated from a simulator or an actual hardware, and Event Feeder plays back the saved event in a simulator or an actual hardware. There are four possible paths of logging and feeding input events as shown in TABLE III. It means we can record events from the simulator and inject them into an actual hardware, and vice versa.

TABLE III. AN EXAMPLE OF EVENT DEVICE CONFIGURATION IN LINUX

	Logging Events	Feeding Events
1	Actual Hardware	Actual Hardware
2	Actual Hardware	Simulator
3	Simulator	Simulator
4	Simulator	Actual Hardware

Additionally, interworkings between the hardware and simulator provide to mobile industries a best-case scenario for supporting remote debugging. The field test engineer with a mobile phone checks communication-related functions in a CDMA service area. Currently communication-related events as well as user input events can be collected through Target Agent. If the engineer finds that the actual phone has flaws in communication-related functionality, he sends the events recorded on site to software developers, who may be working even in non-CDMA service areas, and they play back the events on the simulator in order to fix defects. This is a more efficient way to debug software because it reduces testing and debugging time considerably by avoiding a test executor's manual input.

C. Experiment on device-based event

We implemented the proposed modules for test automation on an open project called OpenMoko. The project aims at delivering Linux-powered phones with a fully open source software stack. The OpenMoko-powered Neo FreeRunner phone is currently being sold, and project contributors are expecting general developers to create more applictions based on OpenMoko. TABLE IV shows a test environment for our experiment. The identical OpenMoko-based mobile software stack is used for both actual hardware and simulator where test applications are installed by Debian packaging system.

TABLE IV. TEST EVIRONMENT

	Specifications brief
Software Stack	Kernel layer: Linux 2.6 Middleware Layer: OpenMoko Application Layer: OpenMoko application suites Opkg: Debian-based packaging system
Autual Hardware	Neo FreeRunner by FIC : - 128MB SDRAM, 256MB flash - 400Mhz ARM processor(Samsung 2442 SOC) - Two 3D Accelerometers(STM LIS302DL)
Simulator	QEMU - Hardware emulation technology Host desktop: Fedora 10

Fig. 5 shows an example of capturing events generated from two accelerometers in Neo FreeRunner, while running the Gwaterpas application leveling tool using an accelerometer. These events are used for testing applications using an accelerometer on the simulator. The events previously generated from the mobile software stack in an actual phone are transferred to the mobile software stack on top of the simulator by the test automation module. The applications on a

simulator are started and executed by the injected events without a tester's manual input.

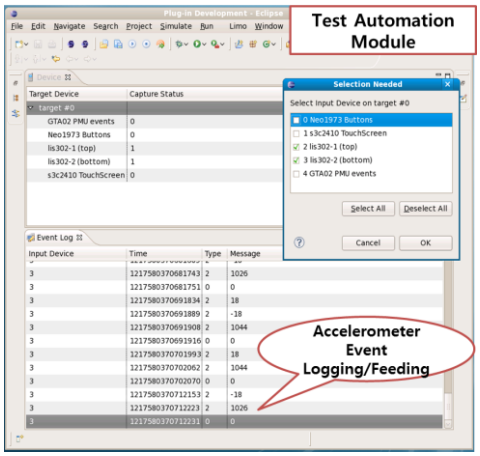

Figure 5. Logging events generated from two accelerometers

Fig. 6 shows application testing based on user input through a touchpad. Fig. 7 shows an example of monitoring an application lifecycle based on D-BUS. The IPC events are generated and saved while running the mobile software stack, and they are used for comparison of actual IPC events generated from the mobile stack during the test. Fig. 8 illustrates accelerometer-driven application testing by using the previously recorded events, as shown in Fig. 5

Figure 6. Applictation testing based on user input event

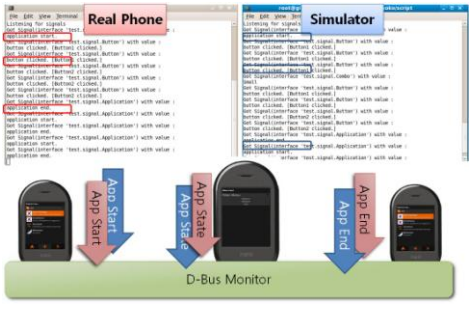

Figure 7. Software module testing based on IPC event

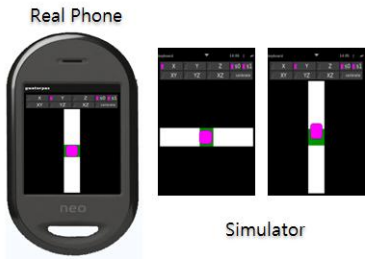

Figure 8. Application tesing based on accelerometer-based event

From above examples, Fig. 6 to Fig. 8, we verify the feasibility of testing the entire mobile software stack by using the events generated from each software layer in the mobile stack.

IV. CONCLUSION

As Linux-powered embedded mobile products increase, it becomes more difficult to do integration testing for a full software stack, which includes the kernel, middleware and applications. Furthermore, the Linux-based mobile platform consists of many open projects developed by different groups. Therefore it is crucial to verify that the added module will not break the harmony of operating the entire system, when a new software module is inserted into the existing mobile platform.

This paper implements a test automation tool of a mobile software stack by using a simulator based on hardware emulation and device events generated in a kernel. Though our approach to handle the input events relies on a Linux driver with a generic event device mechanism, other sensors don't support this mechanism. Challenges remain with hardware emulation for the newly emerging sensors, and support for simulator and device driver should be implemented in order to support the generic event device in the Linux kernel.

REFERENCES

[1] Atif M. Memon, Martha E. Pollack and Mary Lou Soffa, Hierarchical GUI Test Case Generation Using Automated Planning, IEEE Transactions on Software Engineering., vol. 27, no. 2, pp. 144-155, Feb. 2001

[2] Jessica Chen and Suganthan Subramaniam, Specification-based Testing for GUI-based Applications, Software Quality Journal, 10, 205-224, 2002

[3] J. H. Lee, Y. H. Kim, S. J. Kim, "Design and Implementation of a Linux Phone emulator supporting automated application testing", ICCIT 2008, Proc. Vol2, pp.256-259.

[4] J. H. Lee, D. H. Kim, S. J. Kim, C. Ryu, C. H. Lee, "A Test Automation of a full Software Stack on Virtual Hardware-based simulator", ICCIT 2009, Nov. 24-26, Seoul, Korea, pp37-39.

[5] Fabrice Bellard, "QEMU, a Fast and Portable Dynamic Translator," USENIX 2005, proceedings, April 2005, pp41-46.

[6] http://developer.android.com/sdk

[7] http://limofoundation.org

[8] http://wiki.openmoko.org

[9] Vojtech Pavlik, "input.txt", ftp.kernel.org