Generation of Power State Machine for Android Devices

Anh-Tu Bui[1(✉)], Hong-Anh Le[2], and Ninh-Thuan Truong[1]

[1] VNU, University of Engineering and Technology, Hanoi, Vietnam
batu@ictu.edu.vn, thuantn@vnu.edu.vn
[2] Hanoi University of Mining and Geology, Hanoi, Vietnam
lehonganh@humg.edu.vn

Abstract. Power consumption is a major problem on mobile devices. When an application runs, it causes the mobile device to reach a specified state of power consumption. We can determine energy consumption states of mobile devices by analyzing source code of the application. In this paper, we introduce a new approach to modeling energy consumption states due to the impact of Android applications using state machines. The approach takes into account the power states at specific time of the running application. The paper also proposes to construct a finite automata of power states extracted from the source code of the application. We have implemented a plug-in (called PSA) which can be integrated in Android Studio and InteliJ to visualize the finite automata of power states.

Keywords: Formal analysis · Power automata · Power consumption
Mobile devices

1 Introduction

The mobile devices plays an indispensable role in nowadays life. The biggest advantage of these devices is that they are portable thanks to their small size and lightness. Along with the efforts to reduce the size and the weight of devices, their battery capacity must be limited. As a consequence, hardware producers try to find new technologies in order to increase battery capacity, meanwhile software developers try to optimize mobile applications to reduce energy consumption.

Research in estimating the energy usage of mobile devices has investigated in a wide variety of techniques, ranging from specialized hardware, cycle-accurate simulators and operating system level instrumentation. However, these approaches existed their limitations in practice, especially, the ability to early support developers to estimate and calculate energy consumption of apps.

It is clear that making software acting affects strongly to power consuming, such as: software can not able to turn off the screen when people stop using mobile devices, 3G signal transceivers do not turn off when devices are connected Wifi or GPS always acts when devices stay in the same position is the main cause making useless power consuming.

P. Cong Vinh et al. (Eds.): ICCASA 2017/ICTCC 2017, LNICST 217, pp. 48–59, 2018.
https://doi.org/10.1007/978-3-319-77818-1_5

When analyzing the effect of control process by software, we realized that statements in program affects powerfully to working mode of devices, it changed power consumption level of devices in a time unit. This changing appeared when statements required hardware components working such as: when the command *mediaplayer.Start()* appeared, speakers of mobile devices changed from **off** to **on** mode and the power consumer level of devices in **on** mode is higher than **off** mode.

In this paper, we propose an approach to generating Power State Machine (PSM) for Android mobile devices. The approach takes into account the power states at specific time of the running application. The paper also proposes to construct a finite automata of power states extracted from the source code of the application. We have also implemented a plug-in (called PSA) which can be integrated in Android Studio and InteliJ to visualize the finite automata of power states.

The remainder of the paper is structured as follows. Section 2 describes the approach of modeling the power consumption level in mobile devices, using formal methods to model and analyze power consumption level. Section 3 gives several algorithms and the implementation of the support tool (PSA). In Sect. 4, we compared our approach with related works. Finally, we concludes the paper with the main point contribution in Sect. 5.

2 Formal Analysis of Android Applications' Power Consumption

In this paper, we consider a model of one mobile device included only popular hardware components [10]: Audio, GPS, LCD Screen, Wifi, 3G Cellular.

This mobile device is installed Android operating system and the application is run on Android operating system. We just assessed the effect of an individual software to power consumption of device as well, we contemporarily ignored the effect of operating system and other softwares.

To follow the effect of software to the power consumption, we focused on researching the effect of source code to each hardware component, analyzed commands which could affect to the performance of hardware, so that we could detect the different consumption level on these hardware components. And finally, we assessed the whole effects of the application on a mobile device.

2.1 Analysis of the Power State in Mobile Applications for a Hardware Component

Each hardware component had the different active state, such as Audio device had 2 power states: On and Off. We defined a power state for a hardware component as follows.

Definition 1. *A power state of a hardware component is the level of energy consumption in a unit of time, corresponding to the activity levels of the hardware.*

Hardware components are controlled by source code in software. These controlled statements would affect to change active state of hardware, they changed power state of hardware in the application.

Example, whenever users want to turn on the music, the program will carry out the command *Start()* of *MediaPlayer* class, and power state of sound generator Audio will change from **off** into **on**. When carrying out command *Stop* of *MediaPlayer* class, the power state of Audio device will change from **on** into **off**, we describe in Fig. 1.

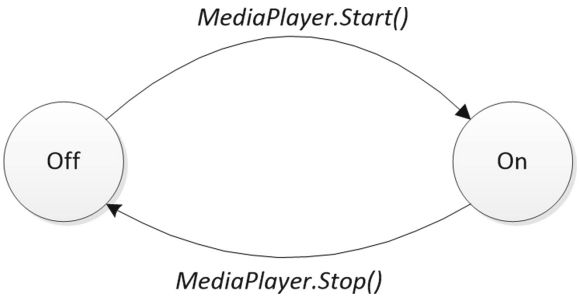

Fig. 1. Power state of Audio device

To model the changing of power state of Audio device, we used a finite automata [4], defined below:

Audio Automata:

$$A_{Audio} = (Q_{Audio}, \Sigma_{Audio}, \delta_{Audio}, q_{0Audio}, F_{Audio})\ [6]$$

where:

$Q_{Audio} = \{off, on\}$
$\Sigma_{Audio} = \{\text{“}Start()\text{”}, \text{“}Stop()\text{”}\}$
$q_{0Audio} = \text{“}off\text{”}$
$F_{Audio} = Q_{Audio}$
δ_{Audio} describles state transition of audio hardware, illustrated in Table 1.

Table 1. State transition table of audio hardware

State-input	Start()	Stop()
Off	On	
On		Off

To do the same actions with other hardware components [2], we give finite automata performing for GPS hardware component as follow:

GPS Automata:

$$A_{GPS} = (Q_{GPS}, \Sigma_{GPS}, \delta_{GPS}, q_{0GPS}, F_{GPS})$$

where:

$Q_{GPS} = \{off, idle, on\}$
$\Sigma_{GPS} = \{$"$PutExtra(String, true)$", "$PutExtra(String, false)$", $RequestLocationUpdates()$"$\}$
$q_{0GPS} =$ "off"
$F_{GPS} = Q_{GPS}$
δ_{GPS} is described by the transition Table 2.

Table 2. State transition table of GPS hardware

State-input	PutExtra(String, true)	PutExtra(String, false)	RequestLocation Updates()
Off	Idle		
Idle		Off	On
On		Off	

LCD Automata:

$$A_{LCD} = (Q_{LCD}, \Sigma_{LCD}, \delta_{LCD}, q_{0LCD}, F_{LCD})$$

where:

$Q_{LCD} = \{off, on\}$
$\Sigma_{LCD} = \{$"$LockNow()$", "$Acquire()$"$\}$
$q_{0LCD} =$ "off"
$F_{LCD} = Q_{LCD}$
δ_{LCD} is describle by a transition Table 3.

Table 3. State transition table of LCD hardware

State-input	LockNow()	Acquire()
Off		On
On	Off	

3G Cellular [8] *Automata:*

$$A_{3GCellular} = (Q_{3GCellular}, \Sigma_{3GCellular}, \delta_{3GCellular}, q_{03GCellular}, F_{3GCellular})$$

where:

$Q_{3GCellular} = \{off, idle, on\}$
$\Sigma_{3GCellular} = \{$"$SetMobileDataEnabled(true)$", "$SetMobileDataEnabled(false)$", $Execute(httpget)$"$\}$
$q_{03GCellular} =$ "off"
$F_{3GCellular} = Q_{3GCellular}$
$\delta_{3GCellular}$ is describle by a transition Table 4.

Table 4. State transition table of 3G cellular hardware

State-input	SetMobileData Enabled(true)	SetMobileData Enabled(false)	Execute(httpget)
Off	Idle		
Idle		Off	Transmitting
Transmitting		Off	

Wifi Automata:

$$A_{Wifi} = (Q_{Wifi}, \Sigma_{Wifi}, \delta_{Wifi}, q_{0Wifi}, F_{Wifi})$$

where:

$Q_{Wifi} = \{off, idle, on\}$
$\Sigma_{Wifi} = \{$"$SetWifiEnabled(false)$", "$SetWifiEnabled(true)$", $Execute(httpget)$"$\}$
$q_{0Wifi} =$ "off"
$F_{Wifi} = Q_{Wifi}$
δ_{Wifi} is describle by a transition Table 5.

Table 5. State transition table of Wifi hardware

State-input	SetWifi Enabled(false)	SetWifi Enabled(true)	Execute(httpget)
Off		Idle	
Idle	Off		Transmitting
Transmitting	Off		

2.2 Analysis Power State in Mobile Applications

Considering all hardware components in mobile devices, we realized that each hardware component had a defined power state in a certain time. Such as, in any

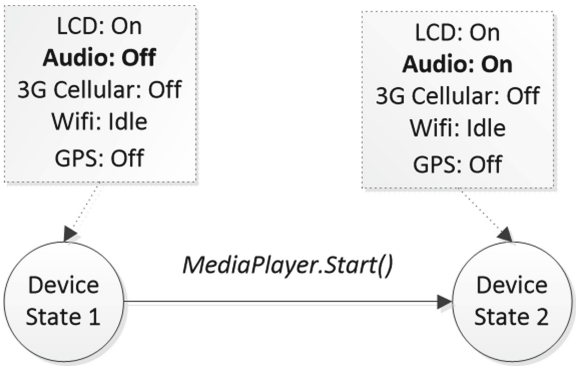

Fig. 2. Power state of mobile application was changed when Audio turn on.

time, LCD Screen was **on**, GPS was **off**, Audio was **on**), Wifi was **idle**, and 3G Cellular was **off**, the power consumption level would be defined and unchanged quantity, called the power state for mobile device.

Definition 2. *A power state of a mobile application (PS_{App}) is a combination of single power states of the hardware components.*

$$PS_{App} = (PS_{LCD}, PS_{GPS}, PS_{Audio}, PS_{wifi}, PS_{3GCellular})$$

The power state of an application in above example (PS_{App}) is:

$$PS_{App} = (on, off, on, idle, off)$$

When there was any hardware component changing state by the effect of program source code, device would change from current power state into other power state. Example, when users turned on the music, power state of device would be describe through Fig. 2.

To specify the whole power states of mobile device, we plan to combine all of states of hardware components, calculate all of the cases that can affect to device. With the recommended method, in the case of having 5 hardware components: LCD, Audio, GPS, 3G, Wifi, we could find out 108 power states of device. These are the power state of mobile device that may appear in applications. To generally model the power state in mobile application, we built a generate automata to perform power states by merging component automata, as below:

$$A = (Q, \Sigma, \delta, q_0, F)$$

Conversion graph is built according to the Algorithm 1.

Algorithm 1. Merge 5 single-automatas

Input:

$A_{audio} = (Q_{audio}, \Sigma_{audio}, \delta_{audio}, q_{0\_audio}, F_{audio})$
$A_{GPS} = (Q_{GPS}, \Sigma_{GPS}, \delta_{GPS}, q_{0\_GPS}, F_{GPS})$
$A_{LCD} = (Q_{LCD}, \Sigma_{LCD}, \delta_{LCD}, q_{0\_LCD}, F_{LCD})$
$A_{cellular} = (Q_{cellular}, \Sigma_{cellular}, \delta_{cellular}, q_{0\_cellular}, F_{cellular})$
$A_{wifi} = (Q_{wifi}, \Sigma_{wifi}, \delta_{wifi}, q_{0\_wifi}, F_{wifi})$

Output:

$A = (Q, \Sigma, \delta, q_0, F)$

1: $Q = \{q | q = (q_{audio}, q_{GPS}, q_{LCD}, q_{cellular}, q_{wifi})\}$
2: $\Sigma = \Sigma_{audio} \bigcup \Sigma_{GPS} \bigcup \Sigma_{LCD} \bigcup \Sigma_{Celluar} \bigcup \Sigma_{wifi}$
3: $q_0 = (q_{0\_audio}, q_{0\_GPS}, q_{0\_LCD}, q_{0\_cellular}, q_{0\_wifi})$
4: $F = Q$ ▷ δ is calculated by following algorithm:
5: **for each** $\{q_{audio} \rightarrow aq_{1\_audio}\} \in \delta_{audio}$ **do**
6: **for each** $(q_{audio}, q_{GPS}, q_{LCD}, q_{cellular}, q_{wifi}) \in Q$ **do**
7: **for each** $(q_{1\_audio}, q_{GPS}, q_{LCD}, q_{cellular}, q_{wifi}) \in Q$ **do**
8: $\delta = \delta \bigcup \{(q_{audio}, q_{GPS}, q_{LCD}, q_{cellular}, q_{wifi}) \rightarrow a(q_{1\_audio}, q_{GPS}, q_{LCD}, q_{cellular}, q_{wifi})\}$
9: **end for**
10: **end for**
11: **end for**

2.3 Optimize Power Consumption Model for Each Specific Application

In the Subsect. 2.2, we introduced general power model for a mobile device performed by finite automata, however, when we work with a particular automata we realize that some of them were not appeared.

They could not appear because there was no control statement in source code of program thus hardware components could not perform corresponding power states. Such as, if there was no Start() command, *on* power states of Audio automata might not appear, this means some of the power state on general automata for device could not be able to appeared.

Therefore, to exactly perform power state for a particular application, we must omit power states that could not be able to appeared. To do this, we have had two ways to perform the combination of component automata models:

- Optimize all component automata before combining them into general automata.
- Combine all component automata and then optimize power states that could not be able to appeared.

In this paper, we choose the first way because it can reduce execution performance. We optimize component automata before combining them into a general automata using the Algorithm 1 presented in Subsect. 2.2. We also used the Algorithm 2 to optimize a single automata in which the input of the algorithm consisted of a general automata and a group of statements chosen from source code that we anticipated that they could affect to power state.

Algorithm 2. Simplify automata by removing useless symbols

Input:

$A = (Q, \Sigma, \delta, q_0, F)$

$\Sigma' = \{a \mid a \text{ is a statement in the program}\}$

Output:

$A' = (Q', \Sigma', \delta', q_0', F')$

$q_0' = q_0$
$Q' = \emptyset$
$newQ = q_0 \bigcup \{q \mid q_0 \rightarrow aq, \{q_0 \rightarrow aq\} \in \delta, \forall a \in \Sigma'\}$
$\delta' = \{q_0 \rightarrow aq, \{q_0 \rightarrow aq\} \in \delta, \forall a \in \Sigma'\}$
while $(Q' \neq newQ)$ **do**
    $Q' = newQ$
    for each $q_1 \in Q'$ **do**
        for each $a \in \Sigma'$ **do**
            if $\{q_1 \rightarrow aq\} \in \delta$ **then**
                $newQ = newQ \bigcup \{q\}$
                $\delta' = \delta' \bigcup \{q_1 \rightarrow aq\}$
            end if
        end for
    end for
end while
$Q' = newQ$
$F' = Q'$

3 Implementation

Recent programming tools, such as Android Studio or IntelliJ allows programmers to analyze commands about syntax, data however it is not supporting programmers in analyzing effect of power states in a program. By inspecting the states, programmers can realize high power consumption states and adjust statement accordingly.

For that reason, we built a Plug-in running on Android Studio and IntelliJ called PSA. PSA used JavaParse to analyze all source code in software projects, it selected statements that could change state of device to optimizing component automata. The overview of PSA Plugin is presented in Fig. 3.

PSA analyze source code of a program and visualize them in a state transition diagram. In the Fig. 4, PSA visualizes Wifi automata for an mobile application.

After optimizing all component automata, PSA combined these automatas into general automata (Fig. 5).

With this state diagram, programmers could observe occurred power state by commands, as a consequences, they could adjust statements in the program to be the most suitable. Each time of adjusting source code, PSA would analyze source code again, create new automata to analyse effects of source code to power state of mobile applications.

Based on assessing the power consumption level in a time unit for every power state of each hardware component [5], we got general power consumption

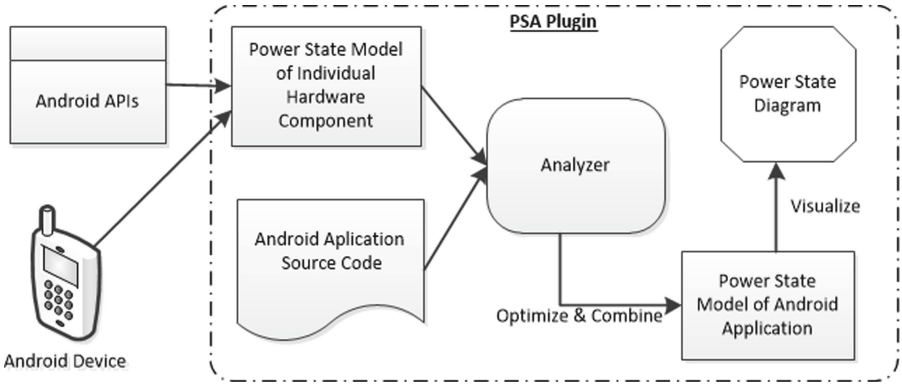

Fig. 3. Overview of PSA Plugin

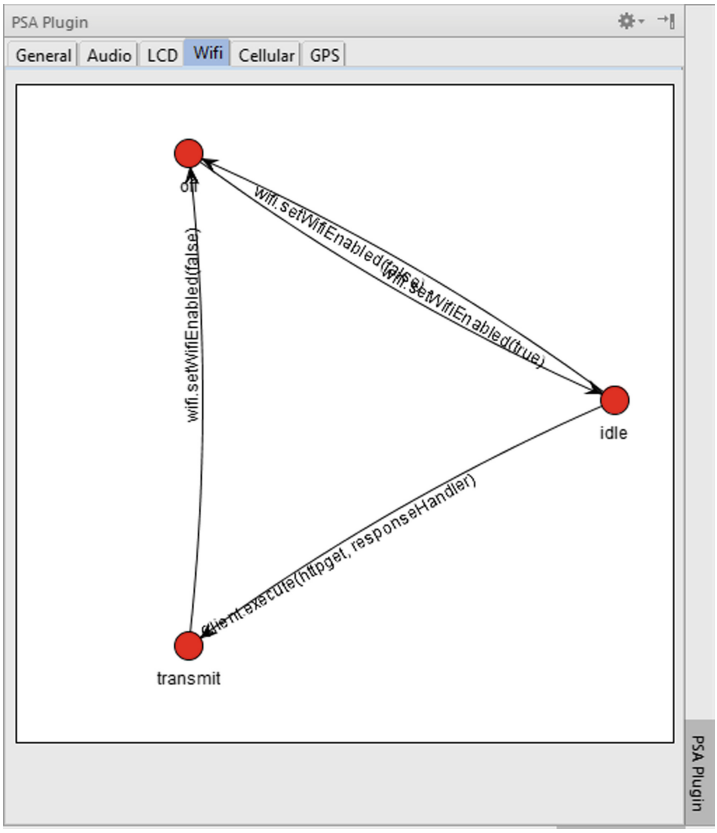

Fig. 4. Visualization of the Wi-fi automata

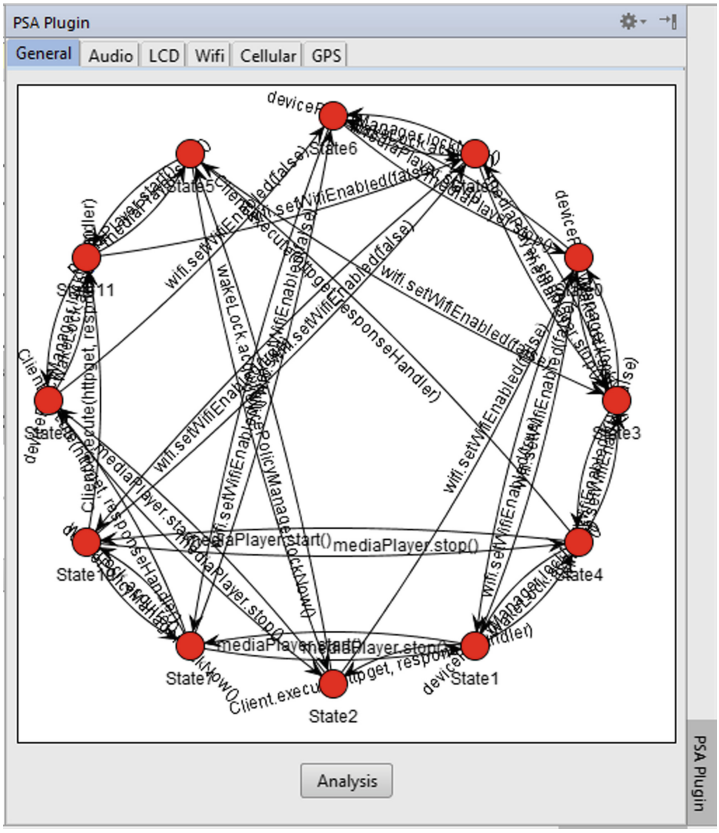

Fig. 5. Visualization of general automata

level in one time unit. This helped programmers know what power state wasted power to quickly adjust the commands, and open the researching trend about power consumption [3] later.

4 Related Work

Nakajima [9] proposed a model-based approach to the representation and analysis of the asynchronous power consumption of Android applications. They introduce a formal model, the power consumption automaton (PCA), show how the PCA is analyzed with existing tools and present some discussions based on their experience.

The paper [7] proposed an approach to estimating power consumption level by analyzing command lines. The paper introduced Elens, a tool used to visualize power consumption level on each command line. This approach permitted calculating power consumption level for command lines in a specific application

however it did not permit analyzing and checking the power constrains in general cases.

Lide Zhang [5] proposed an approach that is both lightweight in terms of its developer requirements and provides fine-grained estimates of energy consumption at the code level. It achieves this using a novel combination of program analysis and per-instruction energy modeling. The approach also provides useful and meaningful feedback to developers that helps them to understand application energy consumption behavior.

Carroll [1] presented a detailed analysis of the power consumption of the Openmoko Neo Freerunner mobile phone. They measure not only overall system power, but the exact breakdown of power consumption by the device's main hardware components. The paper proposed this power breakdown for micro-benchmarks as well as for a number of realistic usage scenarios. These results are validated by overall power measurements of two other devices: the HTC Dream and Google Nexus One. They develop a power model of the Freerunner device and analyse the energy usage and battery lifetime under a number of usage patterns.

In this paper, we work about the generation of Power State Machine for Android devices and then we provide another approach to analysis power consumption of these devices.

5 Conclusion

By assessing the effect of program statements in controlling hardware components of mobile devices, we determine the power states and the corresponding between statements in source code and the changing power states of hardware components. Thus, we built each automata model represented the power states of each hardware component in mobile devices. After finished optimizing these component automata, we combined component automata models to form a general one for a mobile application.

We have built a tool with the aim of supporting programmers on assessing the effect of source code to power consumption of mobile device called PSA. This tool analyse automatically source code and visualize exactly the power state model for each hardware component in mobile application.

The power model enabled programmers to observe the application's power states. We can extend this work by considering the power consumption level and examining the constraints about power consumption over the time.

References

1. Carroll, A., Heiser, G., et al.: An analysis of power consumption in a smartphone. In: USENIX Annual Technical Conference, Boston, MA, vol. 14, p. 21 (2010)
2. Couto, M., Carção, T., Cunha, J., Fernandes, J.P., Saraiva, J.: Detecting anomalous energy consumption in android applications. In: Quintão Pereira, F.M. (ed.) SBLP 2014. LNCS, vol. 8771, pp. 77–91. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11863-5_6

3. Datta, S.K., Bonnet, C., Nikaein, N.: Android power management: current and future trends. In: 2012 First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT), pp. 48–53. IEEE (2012)
4. Grzes, T.N., Solov'ev, V.V.: Minimization of power consumption of finite state machines by splitting their internal states. J. Comput. Syst. Sci. Int. **54**(3), 367–374 (2015)
5. Hao, S., Li, D., Halfond, W.G.J., Govindan, R.: Estimating mobile application energy consumption using program analysis. In: 2013 35th International Conference on Software Engineering (ICSE) (2013)
6. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Boston (2006)
7. Li, D., Hao, S., Halfond, W.G.J., Govindan, R.: Calculating source line level energy information for Android applications. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013, p. 78 (2013)
8. Mendonça, J., Lima, R., Andrade, E., Callou, G.: Assessing performance and energy consumption in mobile applications. In: 2015 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 74–79. IEEE (2015)
9. Nakajima, S.: Model-based power consumption analysis of smartphone applications. In: ACESMB@ MoDELS (2013)
10. Zhang, L., Dick, R.P., Morley Mao, Z., Wang, Z.: Accurate online power estimation and automatic battery behavior based power model generation for smartphones, Ann Arbor