Noname manuscript No.
(will be inserted by the editor)

An approach to modeling and estimating power consumption of mobile applications

Hong Anh Le · Anh Tu Bui · Ninh-Thuan Truong

Received: date / Accepted: date

Abstract Recently, the number of mobile users grow enormously. Even though hardware technologies have been taken to many advantages, which aim at increasing the battery capacity, people are still facing with the problem of battery shortage. An application that runs functionally correct and has a friendly graphic user interfaces still causes users frustrated if it drains the battery. In parallel with increasing the energy storage capability of mobile device, optimizing the source code to reduce power consumption is an emerging topic. This paper presents a new approach to modeling and evaluating power consumption of mobile applications. First, we introduce new definitions of power states and a power consumption automaton (PCA) for a hardware component. In the next step, we propose algorithms to optimize and merge these into an unified automaton. In order to estimate the power consumption amount, the power automaton is refined with time aspect and power coefficients. Finally, we develop a support tool, which is a plug-in for Android studio and IntelliJ for visualizing the power machines and estimating the accumulated power consumption of an application use-case.

Keywords Formal analysis, Power State, Power Automaton, Power Consumption

Hong Anh Le
Hanoi University of Mining and Geology, Hanoi, Vietnam
E-mail: lehonganh@humg.edu.vn

Anh-Tu Bui
VNU, University of Engineering and Technology
E-mail: tuba.di14@vnu.edu.vn

Ninh-Thuan Truong
VNU, University of Engineering and Technology
E-mail: thuantn@vnu.edu.vn

1 Introduction

In recent years, the number of people using mobile devices grow enormously. They are used for not only making phone calls but also entertainment and work. Mobile applications have became more and more complex which handle multimedia files with audio and video, transfer data with cellular and Wifi networks, and navigate using GPS coordinates. Even though hardware technologies have been taken to many advantages which aim at increasing the battery capacity, mobile users are still facing with the problem of battery shortage. For this reason, mobile application developers have to take into account energy consumption level of their applications. An application that runs functionally correct and has a friendly graphic user interfaces still causes users frustrated if it reduces battery remaining quickly. These problems have been identified as energy bugs [15]. One of major reasons that makes mobile software consume much energy is that it contains source codes which access hardware peripherals. Using API functions incorrectly also results in consuming much energy than usual. For instance, the applications still request GPS data while the devices stay in the same position or the screen isnot turned off when the devices are playing audio files and users have no activity for a while. For a specific mobile platform, an application is allowed to actively control power consumption by several methods. For example, in Android operating system, ones can use PowerManager class of Android frameworks where wakelock can be employed. If wakelocks are misused, the applications also might lead to energy leaks.

There are several approaches, which have been dedicated for detecting these bugs in both including design and implementation phases with various techniques. Some work concentrate on improving power consumption ef-

ficiency of a specific hardware of the mobile device. Though, there is a need of new methods for evaluating power consumption.

This article proposes an approach to modeling and estimating energy consumption of an Android application by analyzing its source code. First, we introduce new definitions of power states for hardware components of a mobile device, i.e.., GPS, WIFI, Audio, Cellular, CPU, and display scree. Based on these definitions, we construct a PCA corresponding to each involved hardware component. After that, we propose algorithms to eliminate unnecessary states and merge into an unified PCA. In the final step, we refine the automaton achieved in the modeling step in order to calculate the accumulated power consumption of the application.

In the view point of practice, we develop a plugin, named PCE plugin, for Android Studio and IntelliJ to assist Android developers. This tool is able to extract source codes and formalize power states. It provides users visual diagrams of states transitions and the accumulated power consumption. The tool is helpful for developers to understand the energy effect and optimize the source code.

The remainder of this paper is organized as follows. Section 2 presents a formal approach to modeling power consumption of Android applications. Followed by Section 3, which shows the method to estimate the level that an application consumed. The support tool PCE is described in Section 4. Section 5 compared the achieved results to other related work. Finally, Section 6 concludes the paper and outlines some future work.

2 Modeling power consumption using power state machines

In this section, we propose an approach to modeling source code with respect to power consumption. We only consider the case that the applications work under Android operating system and use the following popular hardware components such as Audio, GPS, Screen, Wifi, and Cellular. We do not take into account the effect of the operating system and other running software.

First, we model the source code to formalize the power consumption on each individual hardware component by a power state machine. Then, we construct a composited automata for all components.

2.1 Modeling power states of a hardware component

Every hardware component has various working states, consuming different amount of power level, which have corresponding power states, e.g., an Audio device has 2 states: On and Off, it has two corresponding power states *On* and *Off*. We define a power state for a hardware component as follows.

Definition 1 <Power state> A power state of p of a hardware component $\mathbb{C}$ represents level of power consumption at a specific working state of the device.

The application may contain source code lines that control or change the states of the hardware device. Hence, it leads to the change of the power states that affects to power consumption of the application. For example, if users want to play a music file, the program might use 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** described in Figure 1.

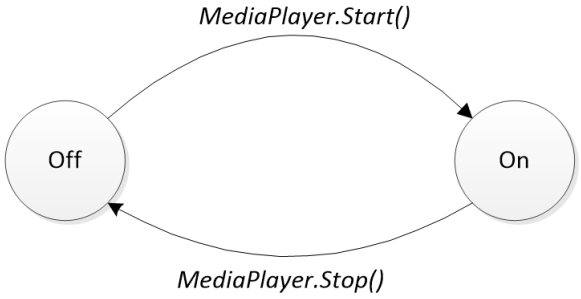

Fig. 1: Power States of Audio device

Definition 2 <Power transition> A power transition t represent the transition from a power state p to another power state p'.

Definition 3 Power consumption of a hardware component is represented by an power consumption automaton (PCA) $\mathcal{A}$ is a 4-tuple $\langle \mathcal{P}, \Sigma, \delta, p_0 \rangle$, where

- $\mathcal{P}$ is a finite set of power states.
- Σ is a set of labels.
- $\delta = P \times \mathcal{L} \rightarrow P$ is a set of power state transitions.
- $q_0 \subseteq \mathcal{P}$ is a finite set of initial power states of the device.

In this paper, we apply these definitions to specific hardware, e.g., Audio, Wifi, Screen, GPS, Cellular equipments, which are involved in a program to analyze the power consumption. Each hardware component is represented by a corresponding automaton as follows.

Audio PCA: The audio component has two power states such as *off* and *on* indicating that whether a program is playing audio. A program can start or stop playing audio using appropriate API statements, e.g. start()/stop() method of *MediaPlayer* class. We define the audio power automata as follows.

$$A_{Audio} = (\mathcal{P}_{Audio}, \Sigma_{Audio}, \delta_{Audio}, q_{0Audio})$$

where:

$\mathcal{P}_{Audio} = \{off, on\}$

$\Sigma_{Audio} = \{\text{“}Start\text{”}, \text{“}Stop\text{”}\}$

$q_{0Audio} = off$

δ_{Audio} describes state transition of audio hardware, illustrated in Figure 2.

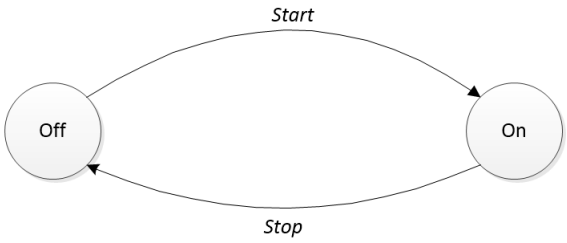

Fig. 2: Audio PCA

GPS PCA: An application tracking location with GPS has three power states including *off* (the application does not turn on GPS features), *active* (the application is receiving GPS data), and *sleep* (GPS feature is turned on but in idle state). In order to turn on/off GPS feature, developers may use *PutExtra)* and *RequestLocationUpdates* API methods of *LocationManager*. We state labels which correspond to statements of Android framework as below

TurnOn = { intent.putExtra(String, true)}
TurnOff ={ intent.putExtra(String, false)}
GetLocation =
{locationManager.requestLocationUpdates()}
Stop = {locationManager.removeUpdates()}
The GPS power automata is defined as follows.

$$A_{GPS} = (\mathcal{P}_{GPS}, \Sigma_{GPS}, \delta_{GPS}, q_{0GPS})$$

where:

$\mathcal{P}_{GPS} = \{off, idle, on\}$

Σ_{GPS} ={*TurnOn, TurnOff, GetLocation,Stop*}

$q_{0GPS} = off$

The state transtitions δ_{GPS} is illustrated in Figure 3.

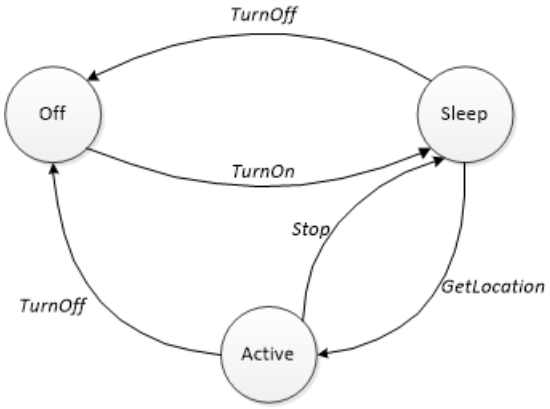

Fig. 3: GPS PCA

Screen PCA: The screen power automata, constructed similarly with Audio component, has two power states such as *off* and *on*. The state of the screen is changed if application uses the wakelock provided by PowerManager class. The transition labels are *Open* and *Lock* respectively defined by statements *wakeLock.acquire()* and *devicePolicyManager.lockNow()*. The screen PCA is construct as follows.

$$A_{Display} = (\mathcal{P}_{Display}, \Sigma_{Display}, \delta_{Display}, q_{0Display})$$

where:

$\mathcal{P}_{Display} = \{off, on\}$

$\Sigma_{Display} = \{Lock, Open\}$

$q_{0Display} = off$

Power state transitions $\delta_{Display}$ is described in Figure 4.

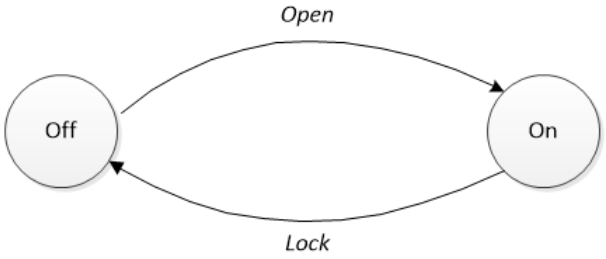

Fig. 4: Screen display PCA

Cellular PCA: An application might use cellular network to send and receive data. If the application is connected to a cellular network for transferring data, the power state is defined as *on* state. In case that the device is turned off, the power state is *off*. Otherwise, it is *idle* state. The power state is changed when

the application invokes *SetMobileDataEnabled* and *Execute* functions of *ConnectivityManager* class provided by Android framework.

$$A_{Cellular} = (\mathcal{P}_{Cellular}, \Sigma_{Cellular}, \delta_{Cellular}, q_{Cellular})$$

where:

$\mathcal{P}_{Cellular} = \{off, sleep, transmit\}$
$\Sigma_{3GCellular} = \{TurnOn, TurnOff, Transfer, Stop\}$
$q_{03GCellular} = off$
The state transitions $\delta_{Cellular}$ is depicted in Figure 5.

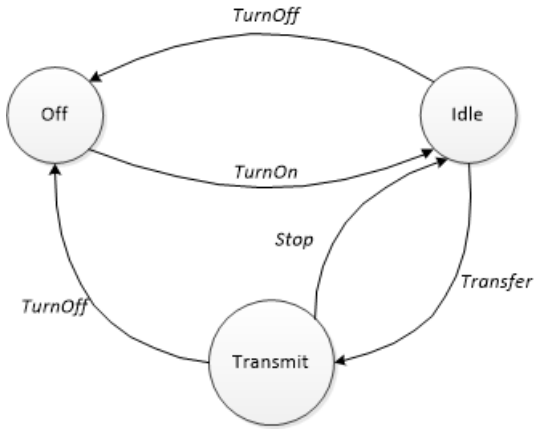

Fig. 5: Cellular PCA

Wifi PCA: If the application executes statements containing API functions such as *SetWifiEnabled*, it can manage the WIFI connection. Hence, the power consumption are affected. In case that the application transfers data, the state is defined as *high-power*. If it is turned on without transfering, the state is *low-power*. We construct the Wifi PCA as follows.

$$A_{Wifi} = (\mathcal{P}_{Wifi}, \Sigma_{Wifi}, \delta_{Wifi}, q_{0Wifi})$$

where:

$\mathcal{P}_{Wifi} = \{off, low - power, high - power\}$
$\Sigma_{Wifi} = \{TurnOn, TurnOff, Stop, Transfer\}$
$q_{0Wifi} = off$
The state transition δ_{Wifi} is described in Figure 9.

2.2 Modeling power states in mobile applications

In section 2.1, we realize hardware components which affects to power consumption and define the respective power automata for ones. In fact, one application can utilize multiple hardware devices for various purposes at the same time. Hence, we need to analyze the power

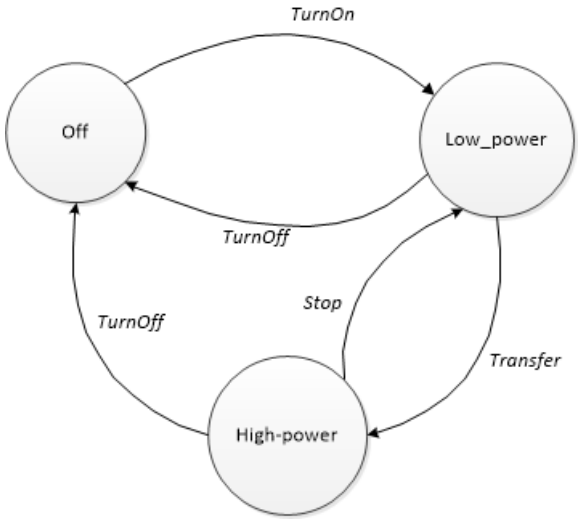

Fig. 6: WIFI PCA

consumption of the whole application. Power state of an application is calculated by combining power states of hardware components that it accesses. We define a power state of an application as follows.

Definition 4 A set of power state of an application ($\mathcal{P}_{App}$) is an union of power states of all hardware components.

$$\mathcal{P}_{App} = \mathcal{P}_{Audio} \cup \mathcal{P}_{LCD} \cup \mathcal{P}_{GPS} \cup \mathcal{P}_{Cell} \cup \mathcal{P}_{Wifi}$$

In case that an application facilitates with hardware components that we already analyzed, a set $\mathcal{P}_{App} = \{Audio_{on}, Audio_{off}, Screen_{on}, Screen_{off}, GPS_{on}, GPS_{off}, GPS_{idle}, Cell_{transmit}, Cell_{off}, Cell_{idle}, Wifi_{high-power}, Wifi_{off}, Wifi_{low-power}\}$. Whenever a hardware component changes its power state because of commands from source code, the power state of the application changes. For example, Figure 7 illustrates the case that if the application just turns on the audio, the audio's power states are change while other states still hold.

In order to formally describe the whole power states of an application, we propose to combine all states of five hardware components, and calculate all of the cases that can affect to device. Based on the approach introduced in section 2.1, where a power states of a hardware component to a power automata, we introduce a composite power automata $\mathcal{A}_{app}$ for the application.

$$A_{app} = (\mathcal{P}, \Sigma, \delta, q_0)$$

We realize that in source codes of a specific application might not contain all statements that change the power states of a hardware component. For this reason, before combining five individual automata, we need to

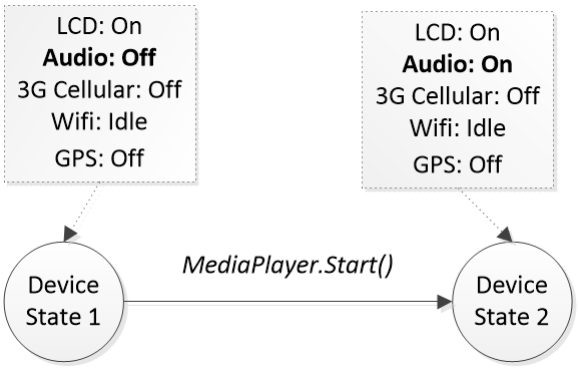

Fig. 7: Power states of mobile application when turning Audio on

optimize the every automaton. The main idea of optimization is removing the power states that does not appear because the corresponding statements were not called. The proposed algorithm iterates all label keywords extracted from source code and searches the path from the initial state. It is detailed in Algorithm 1 for a hardware component.

Algorithm 1 Optimize a PCA by removing omitted states

Input:
$PCA = (\mathcal{P}, l_0, \Sigma, \delta)$
$Commands = \{c \mid c \text{ is a statement in the program}\}$
Output:
$PCA' = (\mathcal{P}', l_0', \Sigma', \delta')$

$l_0' = l_0$
$\mathcal{P}' = \emptyset$
$\Sigma' = \emptyset$
for each $c \in Commands$ **do**
  $a = MapToAction(c)$ where MapToAction is a function mapped from a statement to an action.
  $\Sigma' = \Sigma' \bigcup \{a\}$
end for
$newL = l_0 \bigcup \{l | l_0 \rightarrow al, \{l_0 \rightarrow al\} \in \delta, \forall a \in \Sigma'\}$
$E' = \{l_0 \rightarrow al | \{l_0 \rightarrow al\} \in E, \forall a \in \Sigma'\}$
while $(\mathcal{P}' \neq newL)$ **do**
  $\mathcal{P}' = newL$
  for each $l_1 \in \mathcal{P}'$ **do**
    for each $a \in \Sigma'$ **do**
      if $\{l_1 \rightarrow al\} \in \delta$ **then**
        $newL = newL \bigcup \{l\}$
        $\delta' = \delta' \bigcup \{l_1 \rightarrow al\}$
      end if
    end for
  end for
end while

After eliminating unnecessary power states of each PCA, we introduce an algorithm to merge these into an unified PCA for the application. It is detailed in Algorithm 2. 2.

Algorithm 2 Merge five individual automaton

Input:
$\mathcal{A}_{Audio} = (\mathcal{P}_{Audio}, q_{0Audio}, \Sigma_{Audio}, E_{Audio})$
$\mathcal{A}_{GPS} = (\mathcal{P}_{GPS}, q_{0GPS}, \Sigma_{GPS}, E_{GPS})$
$\mathcal{A}_{Display} = (\mathcal{P}_{Display}, q_{0Display}, \Sigma_{Display}, E_{Display})$
$\mathcal{A}_{Cell} = (\mathcal{P}_{Cell}, q_{0Cell}, \Sigma_{Cell}, E_{Cell})$
$\mathcal{A}_{Wifi} = (\mathcal{P}_{Wifi}, q_{0Wifi}, \Sigma_{Wifi}, E_{Wifi})$
Output:
$\mathcal{A}_{App} = (\mathcal{P}, q_0, \Sigma, E)$

$\mathcal{P} = \{q | q = (q_{Audio}, q_{GPS}, q_{Display}, q_{3G}, q_{Wifi})\}$
$\Sigma = \Sigma_{Audio} \bigcup \Sigma_{GPS} \bigcup \Sigma_{Display} \bigcup \Sigma_{Cell} \bigcup \Sigma_{wifi}$
$q_0 = (q_{0Audio}, q_{0GPS}, q_{0Display}, l_{0Cell}, q_{0Wifi})$
E is calculated by following algorithm:
for each $(\{l_{Audio} \rightarrow al_{1\_Audio}\} \in E_{Audio})$ and $(a \in \mathcal{P}_{audio})$ **do**
  for each $(l_{Audio}, l_{GPS}, l_{Display}, l_{3G}, l_{Wifi}) \in L$ **do**
    for each $(l_{1\_Audio}, l_{GPS}, l_{Display}, l_{3G}, l_{Wifi}) \in L$ **do**
      $E = E \bigcup \{(l_{Audio}, l_{GPS}, l_{Display}, l_{EG}, l_{Wifi}) \rightarrow a(l_{1\_Audio}, l_{GPS}, l_{Display}, l_{3G}, l_{Wifi})\}$
    end for
  end for
end for
Repeat from step 5 to step 11 with GPS, Display, Cellular and Wifi Automaton

3 Estimating power consumption of mobile applications

In order to estimate power consumption of the application state, we need to refine the automaton by adding time and power consumed at a specific power state. Specifically, the power automaton is stated as follows.

$$\mathcal{A}_{app} = (Q, \Sigma, \delta, q_0, T, \mathcal{C})$$

where,

- $\mathcal{P}, \Sigma, \delta, q_0$ is defined in section 2.
- $T = \mathcal{P} \times \mathbb{N}$ is the timing function of the state q and $T(q) = 0, q \in Q$ when the machine leaves the state q.
- $\mathcal{C}$ is coefficient of power consumption state, it depends on specific mobile device. $\mathcal{C} = \{\mathcal{C}_q | q \in \mathcal{P}\}$ where C_q is power consumption coefficient of state q.

To define coefficient at a specific of a hardware component, we reuse the estimation of Zhang Lide *et al.* [16] described in Table 1. The coefficient at a specific state l of the application then can defined as follows.

$$C_l = C_{Audio} + C_{Display} + C_{GPS} + C_{3G} + C_{Wifi}.$$

The accumulated power consumption of an application at a specific time t can be calculated in Equation 1.

$$P_n = \sum_{i=1}^{k} C_l * t_l \tag{1}$$

Table 1: Power coefficients on each state (adapted from [16])

Category	State	Coefficient (min-max)	Category	State	Coefficient (min-max)
$C_{Display}$	off	0	C_{GPS}	off	0
	on	2.40 - 612		sleep	173.55
C_{Wifi}	off	0		active	429.55
	low-power	20	C_{3G}	off	0
	high-power	710 - 758		idle	10
C_{Audio}	off	0		transmit	401 - 570
	on	384.62			

where C_l is coefficient of state l, t_i is the time on state l and $\sum_{i=1}^{k} t_i = n$

4 Support tool for Android studio and Eclipse

Android Studio and IntelliJ are two most popular integrated development environments to develop Android applications. These tools support programmers to analyze source code in term of syntax, it however does not support programmers to analyze the effects to power consumption of the source code that they are developing. Therefore, there is a demand of support tools which are able to visualize the power states modeling and estimate the power consumption of the application. By inspecting the states, programmers can find the energy bugs and adjust source codes accordingly.

Implementing the approach proposed in section 2, we develop a Plug-in, Power Consumption Estimator (PCE), which is fit to Android Studio and IntelliJ. The architecture of PCE is illustrated in Figure 8. PSE Analyzer component integrates JavaParser library [2] to analyze the source code of an Android project. We define power state model of each hardware component as one input of our parser. PCE analyzes code statements and constructs the power consumption automaton.

PCE has two core features:

- Analyzing the source code and visualize the power consumption of each hardware component and the whole application. This feature is described more in Section 4.1
- Estimating the level of power consumption for certain use cases of the application. It is showed in Section 4.2.

4.1 Power Consumption Analysis Tool

To assist the developers in observing the power states of the application, PCE can analyze source code of a program and visualize them in a state transition diagram.

Figure 9 shows the visualization of the Wifi automaton where the application turns on the WIFI to make a request to a web server via http-get protocol.

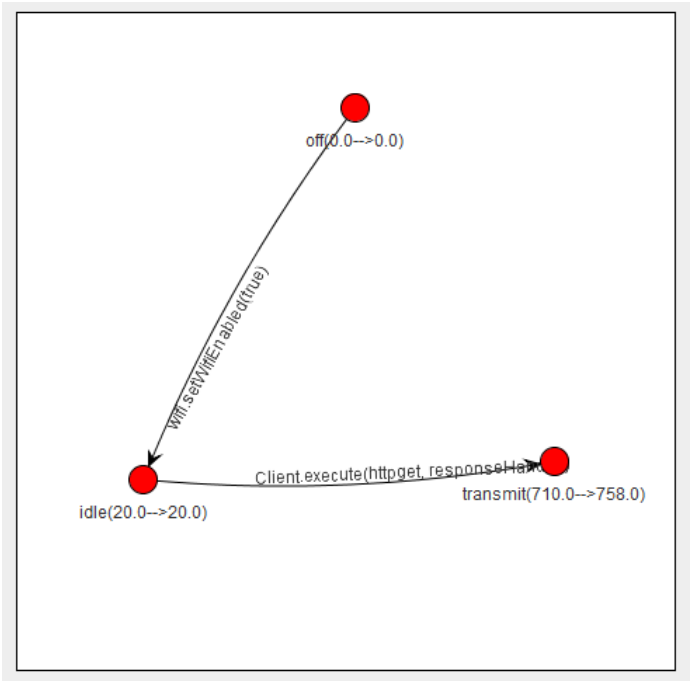

Fig. 9: Visualization of the Wifi automata

Not only support to visualize individual PCA, the PCE tool is able to make visualization of the general automation. Figure 10 shows the PCA of an application that plays audio and uses WIFI to transfer data.

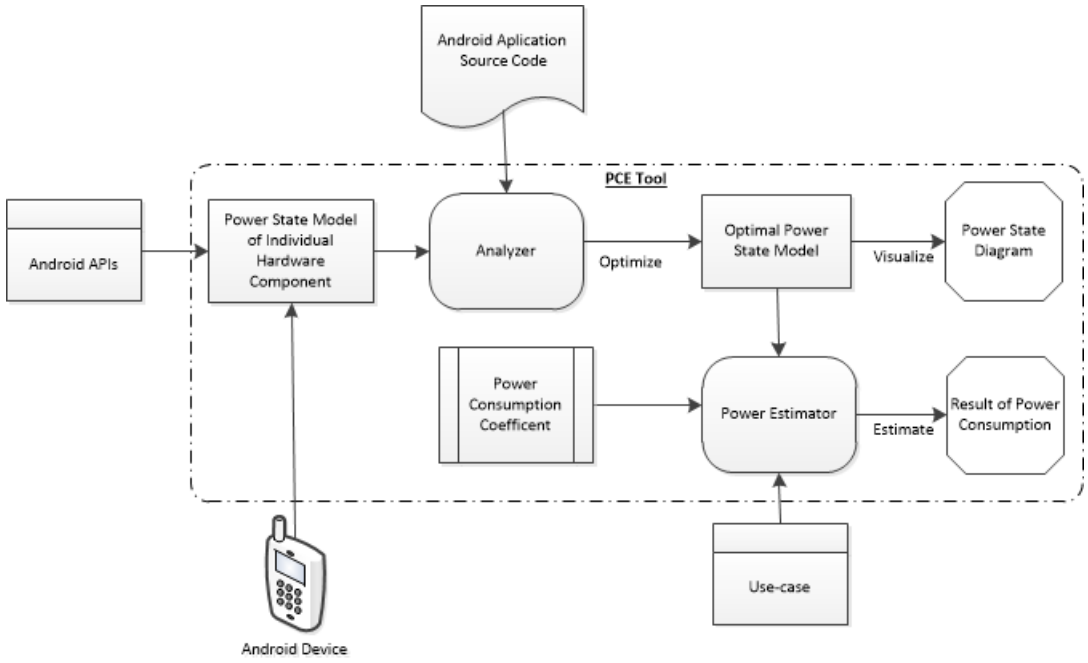

Fig. 8: Architecture of Power Consumption Estimator plug-in

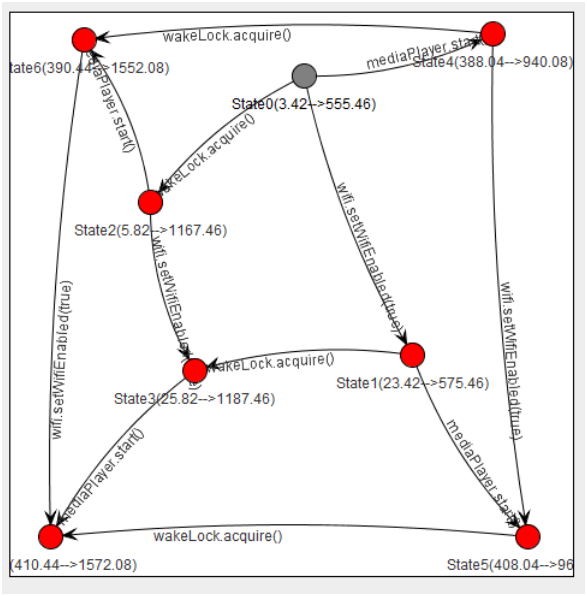

Fig. 10: Visualization of general PCA

Developers can observe the general power consumption automaton generated by the current source code, hence they are able to adjust the statements to optimize the energy usage. Whenever source codes are changed, the tool can reload and update the automaton correspondingly.

4.2 Power Consumption Estimation Tool

The second feature of PCE is to estimate the power consumption of certain use-cases of the application. To to this, developers need to define an use-case by describing a set of user actions. The PCE tool will analyze the defined use-case, then calculate the change of device energy states and return the estimated power consumption.

Syntax for defining the users actions is stated as follows.

$$Component : Action.$$

where *Component* represent for hardware component and *Action* is a power state transition. For example, $Audio : TurnOn$ defines an action to turn on the audio. Moreover, we add additional information to show that the application is executing some tasks in a unit of time.

Figure 11 show the estimated result after running a input use-case of the application. This feature is helpful for developers, if they want to know if the application consumes much energy than expected for a certain use-case.

5 Related Work

Nakajima [14] et al. proposed a model-based approach to the representation and analysis of the asynchronous

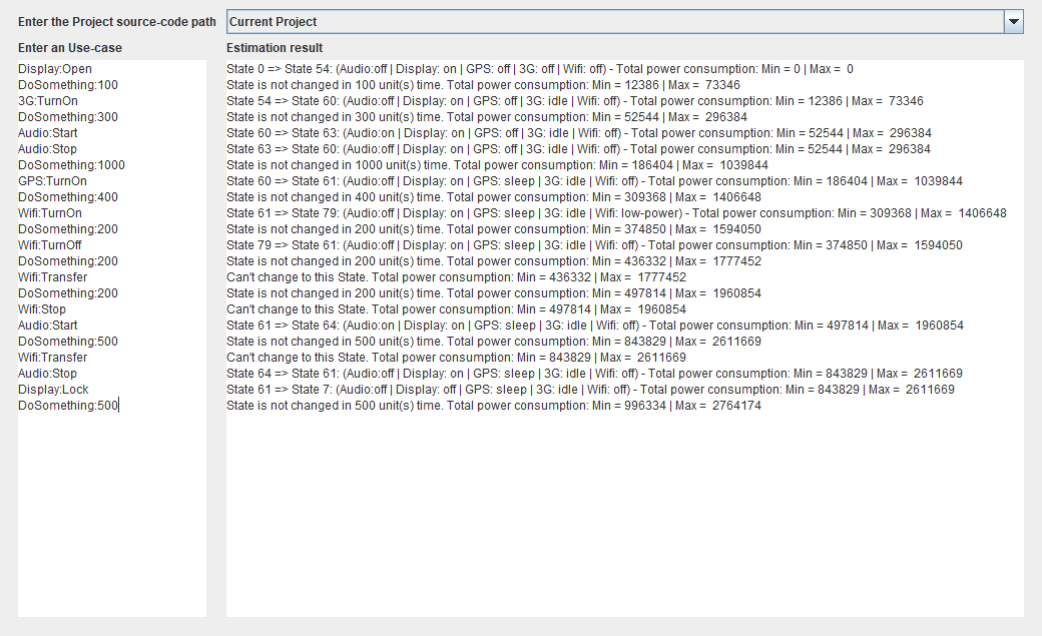

Fig. 11: Result of Power Consumption Estimator

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 [11] 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 [16] 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.

Aaron Carroll [5] 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.

Abhinav Pathak *et al.* [15] presented Eprof whici is a fine-grained energy profiler for smartphone applications. The profiler compare the energy profile running in a conventional computer and a smartphone. It can dectect wakelock bugs and show the location of the bugs.

There are several work dedicated for improving battery efficiency with individual hardware component. Zhenyun Zhuang *et al.* [17] presented an adaptive location sensing framework with design principles including substitution, suppression, etc.. These design principles are implemented as Android middleware and improved the battery life up to 70 percentage. With WIFI connectivity, K. H. Kim *et al.* developed a system, called WiFiSense, employing user mobility information retrieved from low-power sensors. Then the authors proposed some adaptive algorithms to conserve battery power while improving Wi-Fi usage.

Compared to above work, this paper introduces new definitions power consumption automaton and presents an approach to analyzing the power consumption at implementation level. Our work also brings a helpful tool for Android developers.

6 Conclusions

This paper presents an approach to modeling and estimating power consumption of Android applications. We introduce new definitions of power consumption automaton. Based upon these, power consumption automata of hardware components and the general automaton are constructed. Developers realize which parts of source code might lead to energy leaks. In order to estimate the power consumption in certain use-cases of the application, we introduce time-related aspects and power coefficients for each state.

In practice, we develop a plug-in which is suitable for Android studio and IntelliJ. This tool extracts source codes in the project folders, visualizes power automata, and estimates the power consumption for a predefined use-case. The advantages of the proposed approach are providing developers a visual modeling of power consumed of each hardware component based on state machine and the support tool is a plug-in for two most popular development environments for Android communities. Our approach, however, currently works with static source codes and the timer function handles with Natural number. In reality, we need to handle the case of Real number because time aspect is continuous not discrete. One limitation of this paper is that power states extraction is directly based on some certain methods of Android framework classes. In the future, we intend to extend the parser to deal with the application which has more complex structure and use-cases.

Acknowledgments This work is partly supported by the project no. 102.03–2014.40 granted by Vietnam National Foundation for Science and Technology Development (Nafosted).

References

1. Android api. `https://developer.android.com/guide/index.html`.
2. Javaparser. `https://github.com/javaparser/javaparser`.
3. Mediaplayer class. `https://developer.android.com/reference/android/media/MediaPlayer.html`.
4. Ali-Gombe, Aisha and Ahmed, Irfan and Richard III, Golden G and Roussev, Vassil. AspectDroid: Android App Analysis System. *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy*, pages 145–147, 2016.
5. Aaron Carroll, Gernot Heiser, et al. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, volume 14, pages 21–21. Boston, MA, 2010.
6. Marco Couto, Tiago Carcao, Jacome Cunha, Joao Paulo Fernandes, and Joao Saraiva. Detecting anomalous energy consumption in android applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8771 LNCS, 2014.
7. Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pages 48–53. IEEE, 2012.
8. TN Grzes and VV Solov'ev. Minimization of power consumption of finite state machines by splitting their internal states. *Journal of Computer and Systems Sciences International*, 54(3):367–374, 2015.
9. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, Boston, MA, USA, 2006.
10. K. H. Kim, A. W. Min, D. Gupta, P. Mohapatra, and J. Pal Singh. Improving energy efficiency of wi-fi sensing on smartphones. In *2011 Proceedings IEEE INFOCOM*, pages 2930–2938, April 2011.
11. Li, Ding and Hao, Shuai and Halfond, William G. J. and Govindan, Ramesh. Calculating source line level energy information for Android applications. *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, page 78, 2013.
12. Júlio Mendonça, Ricardo Lima, Ermeson Andrade, and Gustavo Callou. Assessing performance and energy consumption in mobile applications. In *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*, pages 74–79. IEEE, 2015.
13. Rahul Murmuria, Jeffrey Medsger, Angelos Stavrou, and Jeffrey M. Voas. Mobile application and device power usage measurements. *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability, SERE 2012*, pages 147–156, 2012.
14. Shin Nakajima. Model-based Power Consumption Analysis of Smartphone Applications. *ACESMB@ MoDELS*, 2013.
15. Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.
16. Lide Zhang, Robert P Dick, Z Morley Mao, Zhaoguang Wang, and Ann Arbor. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones.
17. Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving energy efficiency of location sensing on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 315–330, New York, NY, USA, 2010. ACM.