Static Analysis of Android Apps for Lifecycle Conformance

3 authors:

Samer Zein
Birzeit University
27 PUBLICATIONS **225** CITATIONS

SEE PROFILE

Norsaremah Salleh
International Islamic University Malaysia
74 PUBLICATIONS **1,783** CITATIONS

SEE PROFILE

John Grundy
Monash University (Australia)
753 PUBLICATIONS **11,615** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Process-centred software engineering environments View project

Software Engineering Tools and Software Engineering Environments View project

Static Analysis of Android Apps for Lifecycle Conformance

Samer Zein

Department of Computer Science
Faculty of Engineering and
Technology
Birzeit University, Palstine
szain@birzeit.edu

Norsaremah Salleh

Department of Computer Science
Kulliyyah of Information & Communication
Technology
International Islamic University Malaysia
norsaremah@iium.edu.my

John Grundy

School of Information
Technology
Deakin University, Geelong,
Australia
J.Grundy@deakin.edu.au

Abstract— **Building robust and reliable mobile applications requires the developer to be fully aware of the lifecycle models for mobile applications. During different states of the mobile application lifecycle, such as start-up, running, background etc., various system resources need to be acquired for use and released so that other applications can use them. However, novice and amateur developers, who are a growing fraction in the mobile development community, often find such a task to be non-trivial and complex and limited in support for by existing tools. This paper presents an automated approach based on static code analysis to aid novice developers in managing system resources during different stages of a mobile application's lifecycle. In order to achieve this, we present a software model to encapsulate lifecycle rules for system resources and then create a repository for these resources. In addition, a novel code analysis algorithm is presented to show how Android application source code can be analyzed in order to verify that system resources have been correctly initiated and released. A proof-of-concept software tool known as ALCI has been developed to evaluate our approach. We used ALCI to analyze 10 Android applications and our initial results show that ALCI is effective and successful.**

Keywords-component; lifecycle conformance; static code analysis; automation.

I. INTRODUCTION

Mobile and smart phone adoption is expanding and growing rapidly and millions of mobile applications are available at online stores [1]. This is largely due to the advancement in the hardware industry as modern smartphone devices have faster processors, larger memories, more accurate sensors and faster internet connections. There are several mobile platforms such as iPhone and Windows Phone, and Android platform is one of the most successful [2, 3, 4]. However, novice Android developers and developers who come from different background technologies, such as web and desktop development, face difficulties in developing high quality and reliable Android mobile applications. This is due to Android application development being a nontrivial task and is very different than traditional web and desktop applications[5]. Mobile applications usually have to be developed and deployed in a short time to meet market competition with limited attention being made to testing activities [4, 6].

Although much research in recent years has focused on mobile application testing in areas such as user interface testing, usability and context-awareness (e.g. [3, 7, 8, 9, 10, 11]), little attention has been made to testing mobile application lifecycle conformance[12]. One of the key challenges novice Android developers face is building applications that conform to lifecycle rules [13]. Mobile application development imposes new challenges on novice developers in terms of managing the application lifecycle events correctly to ensure that their applications are reliable [13, 14]. For instance, developers need to have sufficient knowledge about application lifecycle states, conditions and transitions to guarantee their applications do not loose data and manage to free system resources when their application is sent to background or paused. It has been reported that a considerable portion of software bugs in Android applications are related to lifecycle conformance errors [7]. A key to the problem is the ability to correctly manage various system resources as the application goes through different states in its lifecycle. More specifically we are concerned with system resources that when they are not released correctly they can cause other applications to encounter runtime errors [13]. Examples include but are not limited to: cameras, GPS, video, sensors, and network connections.

The Google official web site for Android development [15] is considered to be one of the main resources for Android developers. The website contains information about the lifecycle model and briefly discusses guidelines (lifecycle rules) for developers so as to be aware of when developing Android applications. These guidelines specify how developers should manage system resources during different states of the application lifecycle. However, not only are these lifecycle rules brief and provided in a narrative format, recent studies have shown that the

lifecycle model documented is actually inaccurate and incomplete [13, 16].

This paper presents a novel approach and a toolset that aims to aid novice Android application developers (though it could aid experienced developers too) in building mobile applications that conform to complex and poorly documented application lifecycle rules. Our approach is based on using static analysis of source code to determine whether developers have released system resources correctly or not. We present a new model used to abstract and describe lifecycle rules for different Android application resource utilization, as well as an algorithm of how to check Android source code against compliance with these rules. To validate our method, we developed a software tool as a proof of concept that incorporates both UML model and algorithm. The software tool, we call ALCI (Android Lifecycle Inspector), has been tested against a range of sample Android application source code. Our initial results suggest that our tool can successfully detect if several critical system resources are not being released correctly by developers.

Section 2 provides a review of the problem and motivation for this research. Section 3 describes some of the related work followed by explanation of our approach in Section 4. Section 5 reports the evaluation of our testing tool and section 6 presents discussion of our work. Section 7 presents conclusions from our research

II. BACKGROUND AND MOTIVATION

The *mobile application lifecycle* represents the different states that an application can go through at runtime i.e., when application process is running [17]. When developing applications for traditional desktop operating systems, the application lifecycle is totally transparent to the developer. The operating system takes care of the states of lifecycle to ensure the correct behavior of applications under all cases [13]. By contrast, modern mobile device operating systems such as Android, J2ME and iOS require mobile application developers to be fully aware of specific lifecycle models. Such lifecycle models ensure that mobile application developers build reliable and robust applications with the correct functionality and data integrity over exceptional behaviour, such as the swapping between applications [17, 18, 19, 20].

Operating systems of mobile applications do not save the complete state of an application whenever a state changes in the lifecycle [13]. This is due to the fact that resources are scarce and there is a critical need for efficiency. Developers thus need to have a deep understanding of lifecycle models and build their applications to react correctly to lifecycle events triggered

by the operating systems [13, 17, 20]. Thus, developing mobile applications that conform to lifecycle rules using Android platform is generally a complex task that usually can only be managed and assured by experienced platform developers.

Figure 1 depicts the lifecycle model for Android applications available in Android Developer website [15]. It is important to note that the lifecycle is associated with each graphical user interface (GUI) component that compromises Android application. Such GUI component in Android is known as an **Activity** [18]**.** In Android development, an activity is responsible for presenting a visual user interface for each focused task [4] and an Android application normally consists of one or more activities. Additionally, and unlike other programming paradigms, Android applications are not launched by invoking the main method. Instead, every activity can be treated as independent starting point for the application and has specific sequence of callback methods to start an activity, and another sequence of callback methods to shut it down [21].

Referring to Figure 1, and according to Android Developer's website[15], an activity (at any time) can be in one of the following three states:

- *Resumed*: in this state the activity is in foreground and user can interact with it.
- *Paused*: the activity is partially obscured by another activity, such as dialog. In this state the activity is paused and cannot execute code.
- *Stopped*: the activity is completely hidden by another activity and sent to the background. An example is when user swaps to another application or when user receives a phone call.

Fig1. Illustration of lifecycle model for Android activities **[15].**
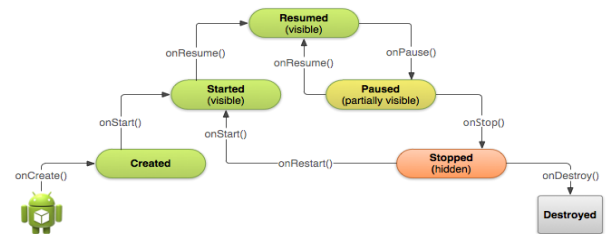
The other states, such as *Created* and *Started,* are not considered as "real" states because an activity moves very quickly through them.
When an activity is started, first the *onCreate()* callback method is called followed by *onStart()* then *onResume()* [19]. After that the activity is in the foreground (*resumed* state) and available to user. When another activity is

started, the former activity moves to background by calling *onPause*() then *onStop*(). If the Android OS decides to kill inactive background activities, *onDestroy*() is the last callback method to be called.

However, recent studies [13, 16] showed that the mobile application lifecycle models suggested by the Android developer website are inaccurate, and in some cases incomplete. More specifically, these studies prove through experiments that at certain circumstances such as extreme low system resources, *onStop*() and *onDestroy*() callback methods are not guaranteed to be called by Android OS before activity is destroyed. This totally contradicts with lifecycle models at Android developer website which suggests that Android developers should use *onStop()* and *onDestroy()* to release system resource, commit changes to database and release running threads [19]. Android applications developed by novice developers who refer to Android developer site would eventually suffer from memory leaks especially if a developer writes code to release system resources in these callback methods [13].

Put simply, and in contrast with Android Developer guidelines, developers should release all resources and save important data using *onPause*() method, as the other methods of *onStop*() and *onDestroy*() are not guaranteed to be called. In this study, we are aware of such problems and our proposed inspection algorithm makes sure that targeted system resources are released at the correct callback method of *onPause().*

III.   RELATED WORK

There is some existing research on test automation tools that are relevant to our present study. Payet et al. [2] realizes the fact that Android programming language features an extended event based-library with dynamic inflation of graphical views from XML layout documents; and that a static analyzer for Android application should realize such features. In their study, they extended the Julia static analyzer to perform formally correct analyses of Android programs. The analysis of Android programs included classcasts, dead code, nullness, and termination analysis. Their tool is based on semantical approach with artificial intelligence through which bugs are found when the tool analysis mechanism has not been able to prove that the analyzed code does not contain bugs.

In a study conducted by Fuchs et al. [8], a tool named SCanDroid is proposed to detect illegal acquisition of security privileges by performing information flow analysis of Android applications and tracking intent (inter-component communication). The tool is also based on constraint-based analysis of source code.

In another study by Lu & Mukhopadhyay [22], a framework for detecting security threats is presented and is based on static analysis techniques that are combined with model-based deductive verification using SMT (Satisfiability Modulo Theories) solvers. The framework can automatically generate an analyzer which is capable of inferring security information about the code. Resulting security information can help developers detect programming errors and permission violation statically.

Almorsy et al. [23] describe a tool that uses formalized signatures defining code patterns for security vulnerability detection. They use static analysis over abstract syntax trees to locate code fragments conforming to these signatures that highlight potential code anti-patterns and thus vulnerability to SQL injection, excessive privilege, poor isolation, inappropriate use of APIs etc. However, their analysis technique does not directly support temporal code relationships e.g. call *release()* method after *allocate()* method across methods.

Franke et al. [13] presents a unit-based test approach for testing conformance of lifecycle dependent properties of mobile applications based on assertions. In their testing approach, the developer has to manually extract lifecycle dependent properties from requirements specification. Then the callback methods are used to test such properties using assertions. Thus having a detailed requirements specification is a centerpiece in this testing approach. This can be problematic because with the rapid nature of mobile application development, detailed specifications are not always available. Another weakness in this testing approach is that the developers have to manually identify callback methods and insert the code for assertions.

## IV.   ALCI ARCHITECTURE AND IMPLEMENTATION

Developing a mobile application that conforms to the Android lifecycle model is a major challenge, particularly for a novice developer [13, 24]. We want to aid novice Android developers in building mobile applications that conform to lifecycle rules by automating the process of inspecting source code against lifecycle rules. If an application reacts correctly to state changes, does not loose data and releases system resources appropriately, then the application is said to conform to the lifecycle rules [13]. More particularly, we focus on system resources that are shared between mobile applications such as Camera, Video, GPS, Microphone, network connections etc. Such resources should be released at certain points (callback methods) when the application is paused or sent to background to make them available to other applications. The importance of correctly managing system resources stems from the fact that failing to do so will cause run time errors and consume other system resources such as battery, CPU and memory space[25].

We first need a formal model to represent the lifecycle rules knowledge base. This is because the current format of Android application lifecycle rules is represented as informal narratives, which cannot be well incorporated into an automated software testing tool. Such a software model should be constructed to represent all lifecycle simple and complex rules. Secondly, lifecycle rules can change over

time: New versions of Android OS are produced regularly [21]. Such versions normally contain changes to old libraries as well as introducing new ones (APIs). Thus the model must incorporate platform versioning. Thirdly, inspecting source code against these rules is necessary: an automated testing tool must be able to automatically detect resource API calls that may be affected by lifecycle rules and match these calls with appropriate lifecycle rules to check if they are being released correctly. Further – as our approach is based on static analysis of source code - a major challenge is imposed because developers can write their code using many different patterns and coding styles. Thus, the tool should be smart enough to recognize such a variety of coding patterns. Finally, current source code analysis tools for Android applications (e.g. [2, 8, 26]) do not inspect their lifecycle conformance.

The main steps of our testing approach are: 1) we define a UML-based model to represent Android system resources along with their lifecycle rules and make them available in a repository; 2) we use a software tool to analyze and inspect Android source code against these formalized lifecycle rules; and 3) the tool produces a report to the developer of notifications of incorrect management of system resources. Figure 2 depicts our proposed testing approach.

Fig. 2 Structure diagram of ALCI
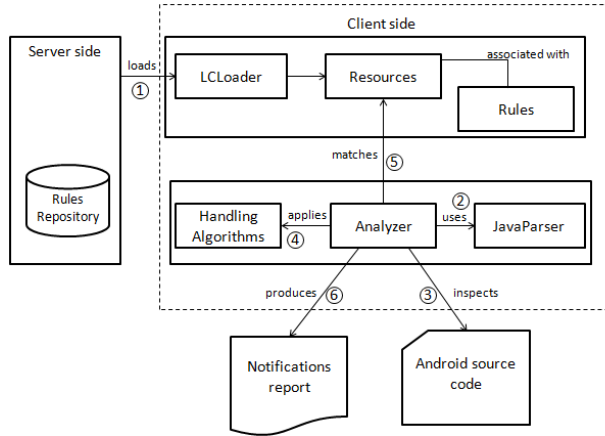
First, the client side of the tool loads system resources and their associated lifecycle rules from repository using the LCLoader component. Second, the analyzer uses JavaParser to parse Android source code. Then using the object model produces by JavaParser, the Analyzer applies handling algorithms that inspects source code against system resources' rules and produces results report. Figure 3 presents our UML model for lifecycle resources and their associated lifecycle rules.

Fig. 3. UML class diagram for lifecycle rules
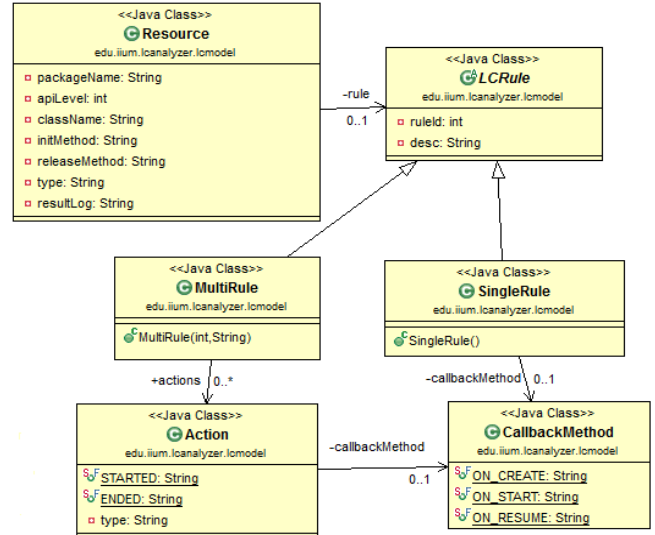
In our lifecycle rules model:

- The Resource class represents the system resources that should be managed by developer which consists of the following attributes and methods:
  o Packagename: name of package for the resource
  o API level: the Android OS level version
  o Class name: the name of class that developer uses to create object in order to use the resource
  o initMethod: the method that initializes the resource
  o releaseMethod: the method that the developer has to call in order to release the resource
  o Type: used to differentiate between various types of resources e.g. Camera, GPS, Microphone, etc.
- LCRule is an abstract class that contains the notification message which will be displayed to developer if he/she did not release a specific resource.
- MultiRule is a subtype of LCRule and it is used when a system resource has to be managed in more than one callback method.
- CallbackMethod represents a single callback method from the Android lifecycle model (e.g. onPause, onStop, onResume).
- SingleRule is used when a resource has to be called at one callback method.
- Action is a class to represent the mode of resource management, mainly initializing (started) and releasing (ended).

A. Code Inspection Algorithm

Source code analysis is the process of analyzing source code to generate useful information for programmers to coordinate their efforts and improve overall productivity [27]. Source code is any static, fully executable description of a software program that can be compiled into an executable form. Static code analysis analyzed the program source code and looks for error patterns without executing the source code [28]. The static code analysis tools would help compare the actual with the expected results.

In our study, our testing tool is based on parsing, analyzing and inspecting Android source code to check whether or not the developer has correctly released system resources. Since the developers can write their code in different styles (patterns), our inspection algorithm should be able to analyze some common coding patterns. Based on our previous experience, we identify two main coding patterns that are commonly used by developers:

  i. First coding pattern is used when a developer calls the *release* method directly inside callback method as in Figure 4 (a).
  ii. Second coding pattern is used when a callback method calls another method which in turn calls the release method as in Figure 4 (b).

Fig. 4 Two common coding patterns
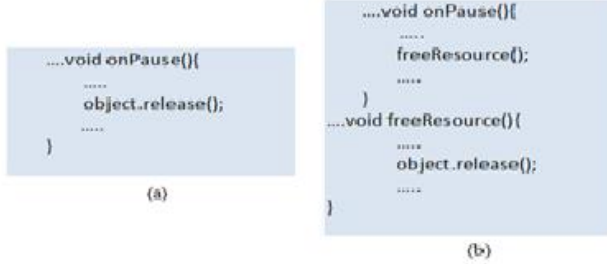
The proposed algorithm for source code analysis can be described in pseudo-code as shown in as follows:

  - **Algorithm Input**: application source code, list of system resources and their associated lifecycle rules.
  - **Algorithm Output**: a list of notifications containing warnings about certain system resources that have not been acquired/ released correctly.

Algorithm basic steps:
  1. Load all system resources information from repository. This includes all resources and their associated lifecycle rules, i.e., when they should be acquired and when they should be released.
  2. Load and parse all "Activities" of the target mobile application. For each Activity, create a list

of all system resources it uses such as Camera, GPS, Thermometer, etc.
  3. For each system resource, analyze source code to verify that this resource has been acquired and released correctly based on the resource rules loaded from repository.
  4. If a system resource has been acquired or released at incorrect lifecycle callback methods, produce a warning message. The warning message should contain the resource name, the exact error in acquiring, releasing, or both, and the correct callback method name for acquiring, releasing or both.

V. EVALUATION

We adopted the evaluation approach applied by Morgado, Paiva and Faria [1] and based our evaluation of ALCI on the principle of seeding bugs into real Android open source applications. This is because there were no other lifecycle testing tools that we could compare our tool against. The evaluation of our Android application lifecycle testing tool was based upon two factors: the ability to successfully detect incorrect and correct releasing of system resources, and performance.

ALCI was evaluated against 10 real open-source Android apps of different sizes and domains, summarized in Table 1. The evaluation process to evaluate the ability of the tool to detect correct and incorrect releasing of system resources consisted of the following three phases:

Phase 1: In this phase, the source code of all applications under test (AUT) was manually checked to make sure they did not contain errors related to lifecycle resources. During this phase, the lists of imports were checked to record which system resources are imported. Then for each of the imported system resource the initialization and release methods are inspected to see if they were called within correct callback methods.

Phase 2: This phase was set to evaluate incorrect release of system resources. During this phase, the source code of applications under test was modified, one by one, by seeding bugs into the main activities and then verifying if the testing tool can successfully detect these errors. Since the tool was based on Camera, GPS and Sensor system resources lifecycle rules, we included THREE (3) incremental testing scenarios for each of the AUT:
  1. First testing scenario: an error was introduced to represent incorrect release of Camera resource and then check whether the tool can correctly detect that error.
  2. In second testing scenario: an additional error was introduced to represent incorrect release for GPS, and then run the tool to see if it can detect both errors (Camera and GPS resources).

3. Third testing scenario: and additional error was introduced to represent incorrect release of Sensor system resource, and then run the tool to see if it can detect all three errors (Camera, GPS, and Sensor).

In this phase, the mechanism of bug seeding was based on manually modifying the source code to import and initiate system resource allocation but not releasing it correctly. Incorrect release of each of the three system resources was done by using a combination of common mistakes observed in novice Android mobile applications: (i) completely omitting the release method (*release()*) from activity ; and by (ii) inserting the release method but at the incorrect callback methods such as *onStop()* and *onDestroy()*. The tool successfully reported incorrect releasing in all cases. A sample notification message showing results for analyzing one of the AUTs can be seen in Figure 5. The notification message shows that the Camera resource has been used but the release command was not found where it should be, in this case, at *onPause()* callback method according to the rule. Additionally, the notification message shows no errors for the other two resources GPS and Sensor as they have not been used yet at that moment. In some cases it would be possible to automatically add source code lines to correct the error, but we have left this for future work.

TABLE 1. 10 open source apps evaluated.

App Name	Type	Description	# lines code main Activity
Cozy DVR	Multimedia	DVR software kit designed for in-car use.	337
Open Camera	Multimedia	A feature-rich camera application.	2559
OSMTracker	Navigation	Journey tracking and mark-up of significant way points.	485
Compass	Navigation	A compass application with a realistic look.	364
FooCam Beta	Multimedia	Takes multiple successive shots with different exposure settings	264
Location Map Viewer	Navigation	Displays geographic information in a map with support for GPX and KML	1002
Music	Multimedia	Plays streams and audio files from the file manager.	144
New Pipe	Multimedia	Lightweight YouTube frontend without the proprietary YouTube-API	356
Sound Recorder	Multimedia	Sound recorder from the MiCode project.	105
Simple Workout Journal	Sports and Health	SWJ is designed for those who know what they want and who are concentrated on results.	923

Phase 3: The objective of the third phase was to evaluate if the tool can detect correct release of system resources. During this phase the idea was to verify if the testing tool can successfully locate the release method and consequently not displaying any error. The tool

successfully detected incorrect resource release faults for all of the seeded faults coding patterns.

Fig. 5 Sample notification message
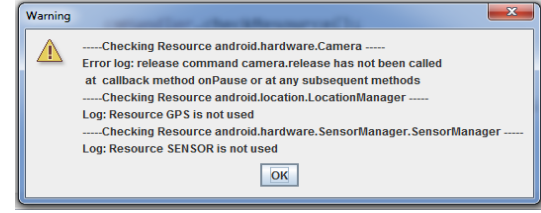
In order to evaluate the performance of the tool [2], execution time was measured in milliseconds. Figure 6 shows the actual execution time required by the testing tool by running it against all AUT.

Fig. 6 A bar graph showing execution time in milliseconds and number of lines of code for AUT
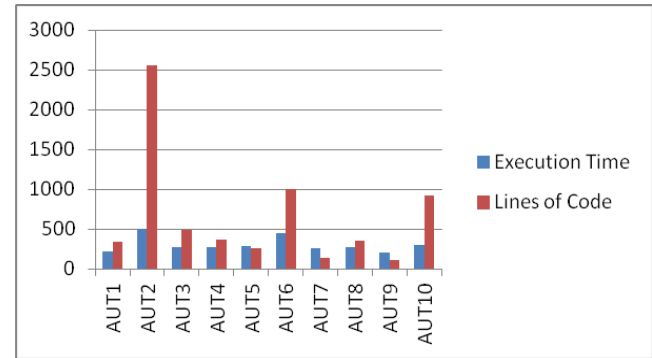
It can be seen from Figure 6 that the time needed for the ALCI to analyze a relatively large mobile application such as Open Camera (AUT2) is very small (about 500 milliseconds).

VI. DISCUSSION

Most studies published in the field of mobile application testing focus on testing areas such as GUI, usability, context awareness, security and compatibility testing (e.g. [4, 7, 12, 29, 30, 31, 32, 33]). Based on the findings from our systematic mapping study [12], we found that there are very few studies that address the importance of lifecycle conformance testing. Furthermore, and to the best of our knowledge, none of the available mobile application testing tools addresses testing of lifecycle conformance. Testing lifecycle conformance of mobile applications is very important to ensure functional correctness of applications as well as data integrity over exceptional behaviors such as swapping-out from one application to another [13, 15].

Our testing approach represents the first attempt to automatically analyze Android application source code to check for errors regarding system resource management during Android mobile application lifecycle transitions. Instead of having lifecycle rules and guidelines being available to developers as narratives; our approach builds a software repository of such rules. The repository can grow over time incorporating new rules and can be available to developers' community as a server side component. In this study, we had proposed ALCI, a client-side, light-weight and expandable software tool that can load lifecycle rules and analyze source code against those rules. ALCI runs as a separate API tool and can be used to analyze mobile applications in relatively small amount of time. In addition, and since that a large portion of mobile application developers are novice [34, 35], ALCI can aid such developers by automatically inspecting their applications. Further, ALCI can identify two common coding patters as discussed in section 5.2. In general, we believe that our proposed testing approach can be generalized and applied to other mobile platforms such as iOS and Windows Phone. This is due to the fact that these two platforms have lifecycle models that are similar to Android OS [36].

However, system resources are only one family of errors that are related to lifecycle conformance in Android applications. There are still other families of errors that ALCI at this stage are unable to detect such as background processes and threads management as well as heavy data persistence management (e.g. writing data to SQLite). Such additional resources should be carefully managed during lifecycle states. Additionally, ALCI produces output as notification messages that are displayed to developers, which, to certain extent can be less practical. It would be potentially more helpful to developers if ALCI could automatically insert corrective code, or at least comments and TODO notifications in the source code. Further limitation of ALCI relates to the coding patterns. At present, ALCI is unable to identify other coding patterns such as when the developers manage system resources in methods that are not part of the Activity class (i.e., classes other than the Activity class itself). Finally, we believe that ALCI can be extended to address other domains of code analysis such as mining Android applications and looking for certain coding patterns of special interest.

A. Threats to Validity

ALCI has been tested on ten real-world Android applications. However, and in order to obtain more accurate results, ALCI has to be tested on other open-sourced Android applications of other domains such as critical applications. Examples of critical applications are but not limited to health, m-government and banking mobile applications. Such diversity in application domains may potentially reveal other coding patterns than those discussed in section 5.2. Further, and since ALCI is currently targeting Android applications, further research could be done on how to generalize approach on other platforms such as iOS and Windows Phone.

VII.   CONCLUSION

In this paper an approach for automatically testing lifecycle conformance for Android mobile applications has been proposed. The testing approach aims at aiding novice Android developers in building mobile applications that conform to lifecycle models when dealing with correctly releasing of system resources such as camera, GPS and sensors. The novelty of our testing approach is that it is the first approach to address mobile apps lifecycle conformance using static code analysis. In this approach, we propose a software model to represent system resources and their associated lifecycle rules and make them available in a repository, an algorithm to analyze source code and detect incorrect releasing cases, and a sample tool as a proof of concept to demonstrate the whole approach. The testing tool known as ALCI is developed as a separate API that can be easily used by novice developers and can efficiently analyze large scale Android applications. The analysis algorithm is capable of detecting two coding patterns applied by most developers. Further, special design patterns such as *Factory Method* and *Strategy* were applied to make the tool extensible and loosely coupled for future expansion. We have tested ALCI tool using ten (10) free and open source Android apps available at *Fossdroid.com* to evaluate the algorithm we used in our testing tool. The results from our experimental evaluation indicate the ALCI tool is able to detect correct and incorrect releasing of system resources. In terms of performance, ALCI is sufficiently efficient to analyze considerably wide range of Android applications in relatively small amount of time.

At the moment, we have not considered other coding patterns that can be used by developers such as having the release commands written in another classes outside the Activity itself as well as other families of lifecycle related errors such as threads management. In future work we intend to extend our analysis algorithm to incorporate such coding patterns. Further, we also intend to enhance the tool itself allowing it to modify source code to correctly release system resources instead of only displaying notifications for developers.

REFERENCES

[1]   Morgado, I.C., A.C. Paiva, and J.P. Faria. *Automated Pattern-Based Testing of Mobile Applications.* in *2014 9th International Conference on the Quality of Information and Communications Technology (QUATIC).* 2014. IEEE.

[2]   Payet, É. and F. Spoto, *Static analysis of Android programs.* Information and Software Technology, 2012. **54**(11): p. 1192-1201.

[3]   Costa, P., A.C. Paiva, and M. Nabuco. *Pattern Based GUI testing for Mobile Applications.* in *2014 9th International Conference on the Quality of Information and Communications Technology (QUATIC).* 2014. IEEE.

[4]   Amalfitano, D., A.R. Fasolino, and P. Tramontana. *A GUI Crawling-Based Technique for Android Mobile Application Testing.* in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on.* 2011.

[5]   Muccini, H., A. Di Francesco, and P. Esposito. *Software testing of mobile applications: Challenges and future research directions.* in *Automation of Software Test (AST), 2012 7th International Workshop on.* 2012.

[6]   Heejin, K., C. Byoungju, and W.E. Wong. *Performance Testing of Mobile Applications at the Unit Test Level.* in *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on.* 2009.

[7]   Amalfitano, D., A.R. Fasolino, P. Tramontana, S. De Carmine, and A.M. Memon. *Using GUI ripping for automated testing of Android applications.* in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on.* 2012.

[8]   Fuchs, A.P., A. Chaudhuri, and J.S. Foster, *Scandroid: Automated security certification of android applications.* Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa, 2009. **2**(3).

[9]   Wen, H.-L., C.-H. Lin, T.-H. Hsieh, and C.-Z. Yang. *PATS: A Parallel GUI Testing Framework for Android Applications.* in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual.* 2015. IEEE.

[10]   Yu, L., W.T. Tsai, Y. Jiang, and J. Gao. *Generating test cases for context-aware applications using bigraphs.* in *Software Security and Reliability (SERE), 2014 Eighth International Conference on.* 2014. IEEE.

[11]   Kronbauer, A.H., C.A.S. Santos, and V. Vieira, *Smartphone applications usability evaluation: a hybrid model and its implementation,* in *Proceedings of the 4th international conference on Human-Centered Software Engineering.* 2012, Springer-Verlag: Toulouse, France. p. 146-163.

[12]   Zein, S., N. Salleh, and J. Grundy, *A Systematic Mapping Study of Mobile Application Testing Techniques.* Journal of Systems and Software: Under review 2015.

[13]   Franke, D., S. Kowalewski, C. Weise, and N. Prakobkosol. *Testing Conformance of Life Cycle Dependent Properties of Mobile Applications.* in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* 2012.

[14]   Zein, S., N. Salleh, and J. Grundy, *Mobile Application Testing in Industrial Contexts: An Exploratory Multiple Case-Study,* in *Intelligent Software Methodologies, Tools and Techniques,* H. Fujita and G. Guizzi, Editors. 2015, Springer International Publishing. p. 30-41.

[15]   *Managing the Activity Life Cycle.* 2017   [cited 2016 May 2016];   Available   from: http://developer.android.com/training/basics/activity-lifecycle/index.html.

[16]   Franke, D., C. Elsemann, and S. Kowalewski. *Reverse Engineering and Testing Service Life Cycles of Mobile Platforms.* in *Database and Expert Systems Applications (DEXA), 2012 23rd International Workshop on.* 2012.

[17]   Lee, W.-M., *Beginning Android 4 application development.* 2012: Wiley. com.

[18]   Murphy, M., *Beginning Android 3.* 2011: Apress.

[19]   *Processes and Application Life Cycle.* 2017   [cited 2016 March];   Available   from: http://developer.android.com/guide/topics/processes/process-lifecycle.html#.

[20]   Haseman, C., *Creating Android Applications: Develop and Design.* 2011: Peachpit Press.

[21]   *Introduction to Android.* 2015   [cited 2015 20/7/2015]; Available from: http://developer.android.com/guide/index.html.

[22]   Lu, Z. and S. Mukhopadhyay. *Model-based static source code analysis of java programs with applications to android security.* in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual.* 2012. IEEE.

[23]   Almorsy, M., J. Grundy, and A.S. Ibrahim. *Automated software architecture security risk analysis using formalized signatures.* in *Proceedings of the 2013 International Conference on Software Engineering.* 2013. IEEE Press.

[24]   Amalfitano, D., A. Fasolino, P. Tramontana, B. Ta, and A. Memon, *MobiGUITAR--A Tool for Automated Model-Based Testing of Mobile Apps.* 2014.

[25]   *Android Sensors Overview.* 2015 25/5/2015]; Available from: http://developer.android.com/guide/topics/sensors/sensors_overview.html.

[26]   *klocwork.* Available from: http://www.klocwork.com/.

[27]   Binkley, D. *Source code analysis: A road map.* in *2007 Future of Software Engineering.* 2007. IEEE Computer Society.

[28]   Louridas, P., *Static code analysis.* Software, IEEE, 2006. **23**(4): p. 58-61.

[29]   Jiang, B., X. Long, and X. Gao. *MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices.* in *Automation of Software Test , 2007. AST '07. Second International Workshop on.* 2007.

[30]   Borys, M. and M. Milosz. *Mobile application usability testing in quasi-real conditions.* in *Human System Interactions (HSI), 2015 8th International Conference on.* 2015. IEEE.

[31]   Amalfitano, D., A.R. Fasolino, P. Tramontana, and N. Amatucci. *Considering Context Events in Event-Based Testing of Mobile Applications.* in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on.* 2013.

[32]   Amalfitano, D., A.R. Fasolino, P. Tramontana, B.D. Ta, and A.M. Memon, *MobiGUITAR: Automated Model-Based Testing of Mobile Apps.* IEEE Software, 2015. **32**(5): p. 53-59.

[33]   Avancini, A. and M. Ceccato. *Security testing of the communication among Android applications.* in *Proceedings of the 8th International Workshop on Automation of Software Test.* 2013. IEEE Press.

[34]   Guo, C., J. Xu, H. Yang, Y. Zeng, and S. Xing. *An automated testing approach for inter-application security in Android.* in *Proceedings of the 9th International Workshop on Automation of Software Test.* 2014. ACM.

[35]   Imparato, G. *A combined technique of GUI ripping and input perturbation testing for Android apps.* in *Proceedings of the 37th International Conference on Software Engineering-Volume 2.* 2015. IEEE Press.

[36]   Starov, O., S. Vilkomir, A. Gorbenko, and V. Kharchenko, *Testing-as-a-Service for Mobile Applications: State-of-the-Art Survey,* in *Dependability Problems of Complex Information Systems.* 2015, Springer. p. 55-71.