2011 18th Working Conference on Reverse Engineering

Reverse Engineering of Mobile Application Lifecycles

Dominik Franke*, Corinna Elsemann† and Stefan Kowalewski‡
Embedded Software Laboratory
Ahornstraße 55
52074 Aachen, Germany
*franke@embedded.rwth-aachen.de
†corinna.elsemann@rwth-aachen.de
‡kowalewski@embedded.rwth-aachen.de

Carsten Weise
IVU Traffic Technologies AG
Borchersstraße 20
52072 Aachen, Germany
http://www.ivu.com
carsten.weise@ivu.com

***Abstract*—In mobile applications, the application lifecycle consists of the process-related states (e.g. *suspended*, *ready*, *running*) and the transitions between them. A faulty or insufficient implementation of the mobile application lifecycle can be the source of many problematic faults, e.g. loss of data. Thus for a software developer, understanding and mastering the mobile application lifecycle is essential for high quality software. In our work with various mobile platforms, we found that the given lifecycle models and corresponding documentation are often inconsistent, incomplete and incorrect. In this paper we present a way to reverse-engineer application lifecycles of mobile platforms by testing. Within a case study we apply the presented concept to three mobile platforms: Android, iOS and Java ME. We further show how developers of mobile applications can use our results to get correct lifecycle models for these platforms.**

***Keywords*-lifecycle; mobile; Android; iOS; Java ME;**

I. Introduction

When talking about application lifecycles this paper does not refer to application lifecycle management in the sense of a software development process (requirements specification, design, implementation, etc.). Instead, *application lifecycle* in this paper refers to the different process-related states of an application during runtime and the transitions between them, as especially found in mobile operating systems. Lifecycles of contemporary mobile applications differ from lifecycles of other systems like desktop applications. Figure 1 sketches the basic application lifecycle of the mobile platform Java ME [1]. The transitions between the different states are not labeled with abstract actions, like start or stop, but with concrete method names. These methods are callback methods executed by the operating system when the application is about to change its state. With this concept the mobile operating system provides the developer a possibility to react on state changes of an application, e.g. turn music off when an incoming call occurs. For this purpose the developer just needs to override the corresponding lifecycle methods. In the following we say a developer *implements the lifecycle (lifecycle implementation)* if he overrides lifecycle callback methods in his application.

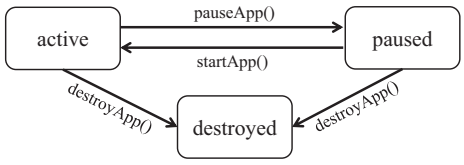

Figure 1. Sketch of Java ME Application Lifecycle

A correct implementation of lifecycles in mobile applications is crucial for high quality software. Imagine the user has typed in a long e-mail text, when an incoming call occurs. The arrival of the incoming call triggers the system to pause the currently focused e-mail application and open the phone application. While closing the e-mail application, the `pauseApp()` method of the e-mail application is called by the system (see Fig. 1). If the user does not store the e-mail text persistently in the `pauseApp()` method, it will be lost. Due to this data loss the user has to retype the whole text when resuming to the e-mail application after the phone call. Another fact that confirms the importance of correctly implemented lifecycles in mobile applications is that lifecycles of mobile interactive devices, like mobile phones or tablets, are heavily stressed. On most mobile devices with a restricted user interface only one application is visible at a time. Since on such devices other resources like CPU and memory are limited too, mobile platforms schedule processes in such a way, that the visible application is the only application in the state *active* (except for not visible active services like music playback). By such a scheduling strategy the currently visible application gets the required resources and stays reactive. For the lifecycles of mobile applications this means, that each time the user switches between two applications (e.g. e-mail and phone application) or returns to the home screen, lifecycle methods are called, since it is not possible to have multiple different mobile applications open and visible at a time, like on a

1095-1350/11 $26.00 © 2011 IEEE
DOI 10.1109/WCRE.2011.42

desktop computer.

Application lifecycles of current mobile platforms are often presented to the developer as a lifecycle model and corresponding documentation. In our work with various mobile platforms, we found that the given information are often incorrect and incomplete. Further, we even found obvious inconsistencies between a given lifecycle model and its documentation. For instance, in the documentation transition sequences are described, that are not traceable in the model. Section II-D describes the found issues with lifecycle documentation in detail.

We started to reverse engineer mobile application lifecycles with a form of dynamic analysis, as, for instance, Systä [2] has done with Java programs or more general with object-oriented programs [3]. Our goal was to derive the real lifecycle model, to be able to implement it correctly and thus react on certain events, like incoming calls, appropriately. A similar approach is also used by Shevertalov and Mancoridis [4] to recover finite state diagrams in order to extract protocols of networked applications. As Briand et al. [5] and Al-Gahmi et al. [6] propose for instrumentation of Java programs and CORBA-based applications, we show how to instrument mobile applications to reverse engineer their lifecycles.

Section II introduces the background of this work. In Section III we describe ways to stimulate these different events and how to derive, from the system reactions, the real application lifecycle model of the platform. Section IV presents the application of this concept to three current mobile platforms: Android, iOS and Java ME. Section V concludes this work.

II. Background

This chapter first introduces the three regarded mobile platforms: Android, iOS and Java ME. To give an idea of mobile application lifecycles, this chapter presents concrete lifecycles of those mobile platforms. Furthermore, some weaknesses of the existing lifecycle representations and documentation of the three platforms are discussed.

A. Android Activity Lifecycles

Android is an operating system and mobile software platform provided by the Open Handset Alliance under the leadership of Google. Android is open-source and deployed under the Apache License. This is a benefit for developers as well as for researchers, since they are able to modify the operating system and log lifecycle-relevant information. Android provides some specific features, that we very briefly introduce now, since they are relevant for the Android application lifecycle[1]. In this paper we use Android 2.2, the currently most widespread Android version. The Android 2.2 specification requires some buttons on Android devices, like *Back*-, *Volume*-, *Home*-, *Call*-, *Cancel*-, *On/Off*- and *Menu*-buttons. The user can overwrite the functions of some buttons, but not of all, e.g. the function of the *Home*-button cannot be overwritten. The Android *status bar* is a bar on the top of the user interface, which is able to show notifications (e.g. SMS). It can be pulled down by the user with a touch gesture over the whole screen. Android refers to the resulting view as *Notification Window*.

An Android application, usually written in Java, can be composed of *Activities* (handling graphical user interfaces) and *Services* (acting in background), among others [7]. In this paper we focus on the activity lifecycle, because of its importance for the user experience of an Android application. During its lifecycle an activity can be in one of four states.

- Usually, an activity is *running* if it interacts with the user in the foreground of the screen. But this is not always the case, as we show in Section IV-A.
- An activity is *paused* if it has lost user focus, but is often still partially visible (e.g. activity is obscured by another activity, which is either transparent or does not cover the full screen). An activity is also paused if the screen locking has been activated, while the activity was in the state running.
- An activity is *stopped* if it is not visible, but still exists in the background. All objects of the activity remain in memory.
- In the state *shut down* no objects of the activity exist in memory.

If an activity resides in the *paused* or *stopped* state, the Android system is able to kill it along with its process, and free some resources.

Figure 2 shows the activity lifecycle model from the Android Developer's Guide. The rectangles represent lifecycle callback methods, which the developer can override to react on a certain state transition. The model also contains possible sequences of callback method calls. For instance, if an activity is started the first time, `onCreate()`, `onStart()` and `onResume()` are called sequentially and the activity is transferred to the state *running*. Two possible states of an activity are given by the model: *running* and *shut down*.

B. iOS Application Lifecycle

Secondly, we examined the iOS application lifecycle. The *iOS* (until June 2010 *iPhone OS*) is a mobile operating system by Apple for iPhone, iPod Touch and iPad. In recent years, next to Android iOS gained also much importance in the market of smartphone operating systems. In contrast to Android, iOS is not open-source and iOS applications are written in Objective-C.

Apple communicates the iOS application lifecycle model

[1]For more details about the following Android components check the Android Developer's Guide on http://developer.android.com/guide/index.html.

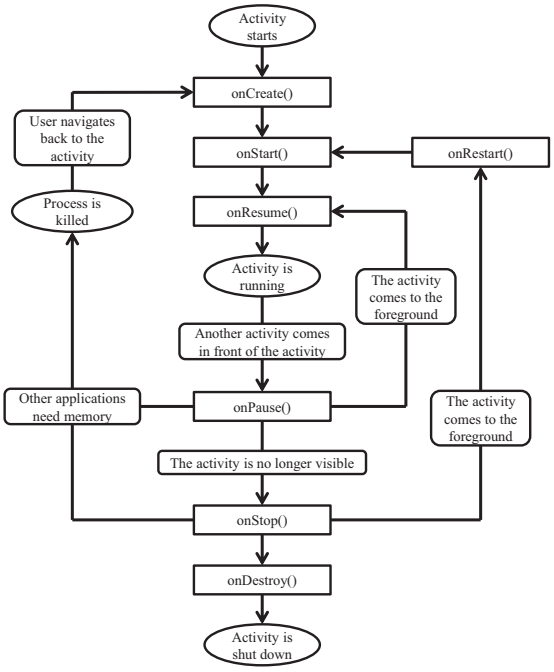

Figure 2. Activity Lifecycle from Android Documentation

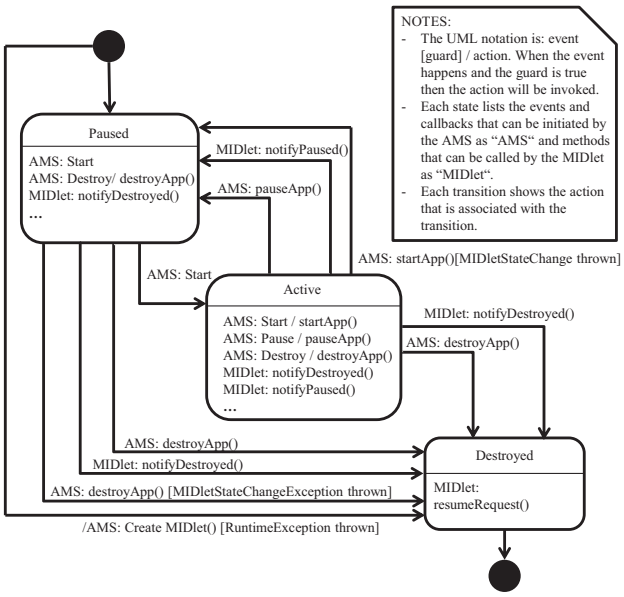

Figure 3. Part of MIDlet lifecycle model from MIDP specification

in the iOS Application Programming Guide[2]. According to the guide, an application for iOS 4, which is the current version, has the following five lifecycle states:

- An application in the state *active* displays contents in the foreground of the screen and receives user and system events.
- In the state *inactive* the application is running in the foreground, but does not receive any events. Nevertheless it can execute code, e.g. when the system prompts the user to react to a SMS message, the active application changes its state to inactive.
- While being in the state *background*, the application executes code, although it is not visible on the screen. Often this state is only entered briefly, while changing to the state suspended.
- If an application is *suspended*, it resides in the background and does not execute any code. In case of lacking resources, the system kills suspended applications without calling any callback method.
- An application in the state *not running* has either not been launched or has been running and was terminated.

With the introduction of multitasking in iOS 4.0 Apple extended the application lifecycle model. The states background and suspended were appended. They are available only in iOS 4.0 and later and on Apple devices that support multitasking.

[2]See http://developer.apple.com/library/ios.

C. Java ME MIDlet Lifecycle

Java Micro Edition (Java ME) is an edition of the Java platform by Sun Microsystems, especially designed for devices with limited resources. The Java technology is characterized by their largely platform independent application programs. Thus, Java ME applications are executable on various devices by different vendors (Nokia, Ericsson, Motorola, ...) with diverse operating systems (Symbian [8], RIM OS, Motorola OS, Windows Mobile,...). In our work we analyze the most common Java ME application class, called *MIDlets*. A MIDlet is an application for mobile devices, which is based on the *Mobile Information Device Profile (MIDP)*.

A part of the basic MIDlet lifecycle is shown in Figure 3 (the complete model can be found in the official MIDP 2.0 specification[3]). Each MIDlet's lifecycle is managed by the *application management software (AMS)*, which calls the callback methods on the system side. During its lifecycle a MIDlet can reside in three different states [1]:

- An *active* MIDlet runs in the foreground on the screen and handles events.
- In the *paused* state the MIDlet is initialized, but does not display any application content on the screen and does not receive any events. The MIDlet is able to execute some code in a background thread, though should not hold any shared resources and should not disturb the active MIDlet.
- A *destroyed* MIDlet has terminated its program execution and has released all of its resources.

[3]See http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html.

Once the user starts the MIDlet, the AMS first instantiates the MIDlet by calling its constructor. After that the MIDlet resides briefly in the *paused* state. The AMS then transfers the MIDlet into the active state and signals this transition by invoking the MIDlet's `startApp()` method. Similarly, its `pauseApp()` and `destroyApp()` methods notify the MIDlet, when it is paused or destroyed, respectively. In contrast to iOS applications, a MIDlet itself is able to initiate certain lifecycle transitions by calling `notifyPaused()` or `notifyDestroyed()`. In Addition, by throwing a `MIDletStateChangeException` a MIDlet is able to prohibit, e.g., a state change from *paused* to *active* (see Fig. 3).

D. Application Lifecycle Problems

Our research reveals several lifecycle-related problems that developers of a mobile application have to face. First, our tests show that the emulators and even more the simulators, provided by the development environments of the mobile platforms, often behave differently compared to a corresponding real device. For instance, the iOS simulator provides only a fraction of functions of the real iPhone. It is possible to simulate a couple of gestures to interact with the touch screen, a low-memory warning, a locked screen, a rotating of the device and pressing the home button. But many events that affect the application lifecycle cannot be simulated and thus are hard to test for the application developer. For instance, it is not possible to simulate an incoming phone call or an empty battery. Additionally, CPU capacity and available memory space of the virtual device equals the CPU capacity and available memory space of the executing Mac. Thus, the iOS simulator is unable to be a substitute for testing with a real iOS device.

Furthermore, the lifecycle representations published in the documentation and specifications of Android, iOS and Java ME vary strongly among the platforms. While the MIDlet lifecycle model is a comparatively formal and well defined state chart (see Fig. 3), the lifecycle representations of Android and iOS have informal syntax and are incomplete. The Android Developer's Guide shows the activity lifecycle model as a whole (see Fig. 2), whereas the iOS Application Programming Guide gives no graphical overview model for the overall iOS application lifecycle. In the iOS Application Programming Guide the developer only finds descriptions of partial lifecycle models and some exemplary events that trigger certain lifecycle transitions.

Further, we found incompletion and inconsistencies in the lifecycle representations. Regarding the Android platform, both, the activity lifecycle and the service lifecycle documentation in the Android Developer's Guide, offer several weaknesses. For instance, the state definition of the activity lifecycle in the documentation does not define the state *shut down*, but the graphical representation of the lifecycle model comprises the two states states *running* and *shut down* (see Fig. 2). The states *paused* and *stopped* are not included in the graphical representation, whereas they are existent in the textual documentation. Moreover the sequence of the callback method calls `onStart()` immediately followed by `onStop()` is not displayed in the graphical model, although the documentation text as well as our tests show that this sequence is possible. Additionally there is an imprecise definition of the state *running* given in the Android documentation. According to the Developer's Guide an activity in this state "is in the foreground of the screen and has user focus". But a running activity is not always in the focus of user interactions. We show this in Section IV-A.

Although the MIDlet lifecycle specification is comparably formal, not all transitions follow the defined syntax. E.g., a transition from the *start* state to *destroyed* is labeled with */ AMS: Create MIDlet() [RuntimeException thrown]* (see Fig. 3). This does not fit the notation *event [guard] / action*. Thus, it is not clear if *[RuntimeException thrown]* is the guard of the transition and what *AMS: CreateMIDlet()* is.

Due to the differing, ambiguous, incomplete and inconsistent representation of the lifecycle models, the developers might run into problems understanding and implementing the mobile application lifecycles properly. In this paper we introduce a remedy to extract complete lifecycle models by reverse engineering, with the objective of bringing clarity to the developers.

III. Reverse Engineering Lifecycles

In this section, we present how to reverse engineer application lifecycles of mobile applications. For this we assume that the developer creates a test application, which might be a minimal *Hello, world!* application. We explain the technique in general and apply it in the next section to real mobile platforms. The method consists of four steps.

Full Implementation of Lifecycle: As a first step the developer has to implement the whole application lifecycle. He has to overwrite each callback method that is called by the platform as a result of lifecycle state changes (e.g. pausing an active application). Taking the lifecycle model shown in Figure 1 the developer had to overwrite the methods `pauseApp()`, `startApp()` and `destroyApp()`. Sometimes not all methods, which are relevant to the lifecycle, are presented within a given lifecycle model. Additionally, some methods are called during lifecycle changes only under certain conditions. For instance, the Android platform calls methods like `onSaveInstanceState()` only when an application is transferred (without user-intention) to a state, where it might get killed by the Android system. `onSaveInstanceState()` stores GUI status data, like position of the cursor within a text view element. It is called before Android's `onPause()` method. But since this method is only called under certain circumstances, it is not treaded by the documentation as a part of the

Android application lifecycle. Thus, during this first step the developer has to find out, which callback methods are relevant to his application and overwrite them in his test application.

Log Injection: In the second step the developer has to add logging functionality to all overwritten methods. Each time one of the callback methods is called, the name of the callback method and the name of the current application shall be logged. Further, if a sequence of callback methods is called, the order shall be obvious. This can be done either by logging additionally timestamps or by having one central logger module, to which each software component reports. This gets important if one application is started by another application (see Section IV-A, third step). For the model given in Figure 1 a possible logger output is given in Figure 4. In this example application App1 is active and another application, called App2, shall be started. Then first App1's `pauseApp()`-method is called, second App2's `startApp()`-method and last App1's `destroyApp()`-method.

Transition-Trigger Detection: Each mobile platform has a finite amount of triggers, that cause a different reaction of an application. For instance, an incoming SMS affects an application different than an incoming call. While an incoming call usually stops a running application and gives the user the opportunity to react on the call, by accepting or declining, an incoming SMS is often just denoted by an acoustic or vibrating signal in the background, not interfering with an active application. The goal of this step is to get a catalog of triggers, that cause different reactions of the mobile platform. This can be done by black-box testing the mobile platform [9]. The developer defines a test case, e.g. *incoming call occurs while the test application is active*, and executes the test on a real mobile device. With the corresponding logger messages, printed by the test application, he can monitor which callback methods are called by the mobile platform during this test.

We prefer real devices, as emulators or simulators do not always behave like corresponding real devices. Some disadvantages of emulators and simulators are given in Section II-D. But emulators do also allow different actions, that are hard to test with real devices. For instance in the Android emulator you can limit the amount of available memory. This way you can trigger the platform to kill a

Time (ms)	Application	Message
...	...	...
10002	App1	pauseApp() called.
10039	App2	startApp() called.
10048	App1	destroyApp() called.
...	...	...

Figure 4. Logger Example with two Applications

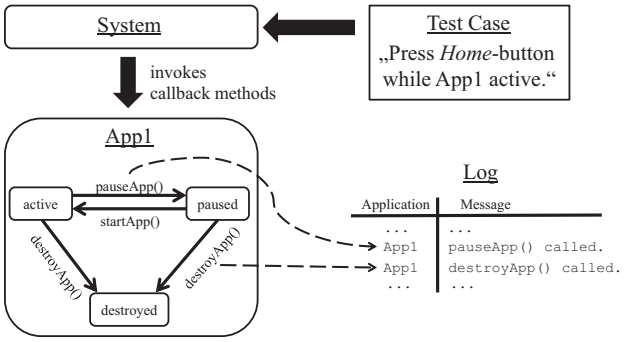

Figure 5. Logging during Test Execution

test application due to insufficient resources. This test case is hard to execute on a real device, since recent devices often have large amounts of available memory, e.g. Google Nexus S has 512 MB of RAM, which cannot be decreased for test purpose.

Figure 5 sketches a possible scenario. The test case consists of pressing the *Home*-button, while application App1 is active. After execution of this test the developer can see, that pressing the *Home*-button, triggers App1 to be paused and then to be destroyed, by calling first `pauseApp()` followed by `destroyApp()`. After a certain number of test cases the developer will notice, that no new sequences of callback method calls accrue. In this case the catalog is complete. Here it is important that the developer finds out all possible lifecycle methods, that are called by the mobile platform. For instance, if the method `onSaveInstanceState()` would exist in the model used in Figure 5 and the developer did not overwrite it, he wouldn't notice that in some test cases this method was called, and thereby the state of the application was stored, and in other cases not. Further, often nearly similar events might cause different reactions: In Android it is a difference, if you exit an application via *Home*- or *Back*-button, although on many user interfaces of different applications the reaction is the same, like returning to the home screen. Therefore, we advise developers to check the following sources for lifecycle callback methods:

1) application lifecycle model, if available
2) documentation of the application lifecycle
3) source code documentation of the application class(es)

In general it is hard to determine, when a catalog of triggers and consequences is complete. At least the developer has to cover the most common triggers, that might occur during the usage of a mobile application, like incoming phone call, accepting the phone call, declining the call, receiving SMS, interference by another application (e.g. alarm-clock) and so on. In the following case study we provide a number of triggers, which can be used as a basis for other platforms. Mobile devices of the same class (e.g. smartphones, tablets) have many equal triggers. Differences

occur especially if one mobile platform supports additional keys, like an emergency or flashlight key. On Android even rotating a device triggers multiple lifecycle methods to be called. The developer has to investigate if and how such platform specific triggers impact the application lifecycle.

Application Lifecycle Rebuild: Based on the collected information the developer is able to derive an application lifecycle model. If, for instance, `startApp()` would never be followed by `destroyApp()`, then the corresponding lifecycle model would not have a transition labeled `destroyApp()` leaving a state X, with an incoming transition to X labeled `startApp()`. So in Figure 1 the transition connecting the states *active* and *destroyed* would have to be deleted.

By this approach the developer can also find new states of an application, that are neither mentioned by the model, nor communicated comprehensible by the documentation. For example, in Android a service can be running (e.g. music playback service) and bound to an activity (e.g. music player GUI) or running without being bound. For the developer it is important if the service is bound or unbound, since in both cases different transitions are possible. If a service is bound, other transition sequences are possible, than if it is unbound. Further an unbound service might be rather shut down due to a lack of resources, than a bound service. In the resulting model such states have to be added.

Next to rebuilding the model the user can also derive various other properties of the application lifecycle from the test results. For instance, he can determine if an application is always visible, when it is in the state *running*. Such information help the developer to assign a running application to a certain state. Further, if an application is running and not visible, resource-consuming interface components like OpenGL computations can be paused. Another important implication from the test results is to deduce if and which callback methods are called if an application gets killed by the system, e.g. due to a lack of resources. Some of the reviewed mobile platforms still give the developer the possibility to react on a kill command by the system and others do not. If a mobile platform does not call any lifecycle callback method when the application gets killed, the user has no chance to react on the killing and data might get lost.

To get more concrete, we apply the presented approach in the following section to three different mobile platforms. The whole procedure has to be done once for a mobile platform. After the model has been rebuilt and ambiguities in the documentation have been clarified, the results apply to each mobile application on this platform, since all applications implement the lifecycle provided by the platform. Furthermore, the mobile platform has the control over the lifecycle by calling the callback methods. The platforms behave equal for all applications of the same type (e.g. activities). Of course there might be different active components available on a mobile platform, each having its own lifecycle, like activities and services. Then this procedure has to be done once for each active component the developer is interested in.

IV. Case Study

This section presents the application of the introduced reverse engineering mobile applications procedure to the three mobile platforms Android, iOS and Java ME. The structure of each subsection aligns to the four steps given in Section III.

A. Reverse Engineering the Android Activity Lifecycle

On the Android platform multiple active components like activities and services are available. In the following we present the reverse engineering of the Android activity lifecycle (see Section II-A).

Full Implementation of Lifecycle: In a test application we create an activity and overwrite the following methods:

- `onCreate()` (abbr. `Cre()`)
- `onStart()` (abbr. `Sta()`)
- `onResume()` (abbr. `Resu()`)
- `onPause()` (abbr. `Pau()`)
- `onStop()` (abbr. `Sto()`)
- `onRestart()` (abbr. `Rest()`)
- `onDestroy()` (abbr. `Des()`)
- `onSaveInstanceState()` (abbr. `SavIn()`)
- `onRestoreInstanceState()` (abbr. `ResIn()`)

Most of these callback methods are given by the official Android lifecycle model, except the last two. The methods `onSaveInstanceState()` and `onRestoreInstanceState()` are not given in the official lifecycle model. But as these methods are called on state changes of the activity, we take them under consideration during our tests. We want to know if and under which circumstances the developer can use these methods to react on state changes of the application.

Log Injection: Android provides an integrated `Log` class in the `android.util` package. This class provides multiple features to log information. In our tests we log data by the following command `Log.d(TAG, MESSAGE)`. The log information are stored with a tag string and a message string. The tag string must identify the current activity unambiguously. The message string has to identify the current overwritten method unambiguous. We give an example for the log entry in the `onCreate()` method:

```
Log.d("activity1",
      "onCreate() called.");
```

activity1 is the name of the activity that we start in our test application at first. The tags in other activities have simply incremented numbers in the string. You need further activities in your test application, for instance, if you want to know which lifecycle methods are called if another activity is opened from within the current activity (see

Table I
TRIGGER CATALOG FOR ANDROID 2.2 PLATFORM

Trigger	Reaction
(1) start application by clicking on the application item on the home screen	`Cre()`, `Sta()`, `Resu()`
(2) after (1) receive an incoming call	`Pau()`, `Sto()`
(3) after (2) accept the incoming call	-
(4) after (2) decline the incoming call	`Rest()`, `Sta()`, `Resu()`
(5) after (3) end the current call	`Rest()`, `Sta()`, `Resu()`
(6) after (1) receive incoming SMS	-
(7) after (6) open SMS through *Notification Window*	`Pau()`, `Sto()`
(8) after (7) return from SMS application back to test application by pressing the *Back*-button	`Rest()`, `Sta()`, `Resu()`
(9) after (1) press *Back*-button	`Pau()`, `Sto()`, `Des()`
(10) after (1) press *Home*-button	`Pau()`, `Sto()`
(11) after (10) start test application again by clicking the application icon on the home screen	`Rest()`, `Sta()`, `Resu()`
(12) after (1) press *Call*-button	`Pau()`, `Sto()`
(13) after (12) return from phone application back to test application by pressing the *Back*-button	`Rest()`, `Sta()`, `Resu()`
(14) after (1) press *Cancel*-button	`Pau()`
(15) after (1) activate screen locking by shortly pressing the *On/Off*-button	`Pau()`
(16) after (15) press *Menu*-/*Home*-/*Call*- or *Cancel*-button	`Resu()`
(17) after (16) unlock screen locking or pull *status bar* over the whole visible test application	-
(18) start test application externally (e.g. via USB) while screen locking is active	`Cre()`, `Sta()`, `Resu()`, `Pau()`
(19) after (1) change device orientation (e.g. from vertical to horizontal)	`Pau()`, `Sto()`, `Des()`, `Cre()`, `Sta()`, `Resu()`
(20) after (15) change device orientation (e.g. from vertical to horizontal)	`Sto()`, `Des()`, `Cre()`, `Sta()`, `Resu()`, `Pau()`
(21) after (10) change device orientation (e.g. from vertical to horizontal) and start test application by clicking the application icon on the home screen	`Des()`, `Cre()`, `Sta()`, `Resu()`
(22) after (1) change volume by using *Volume*-buttons	-
(23) after (1) alarm clock rings	`Pau()`
(24) after (23) discard or snooze alarm clock	`Resu()`
(25) after (1) devices shuts down due to low battery	`Pau()` `onStop()`
(26) after (1) start a sub-activity that overlays the main activity only partially (e.g. small *Dialog*-activity), then press the *Home*-button and return to test application by clicking the application icon on the home screen, then press the *Home*-button again	`Pau()`, `Sto()`, `Rest()`, `Sta()`, `Sto()`

example in Figure 4). The message string is adjusted for each overwritten method, containing the method name of the surrounding method.

Transition Trigger Detection: Table I lists in the left column the various triggers that we identified for the Android platform and in the right column the lifecycle reactions of the main test activity (not of any sub-activity). Due to space limitations we use the abbreviations given above. The start configuration for each of the tests consists of the device after a fresh startup, with no services or applications started but the default system applications and services, and navigated to the home screen, on which an application icon of the test application is visible. In the test results we do not list calls of the methods `onSaveInstanceState()` and `onRestoreInstanceState()`, since their behavior during all tests corresponded to the specified behavior in the documentation. And as predicted in the documentation these methods are only called under certain circumstances, given in the Android documentation as well. For this reason we also do not integrate these methods in the resulting lifecycle model. During all tests all buttons had the default functionality, given by the system. We did not overwrite any functionality of the buttons, e.g. *Back*- and *Volume*-buttons.

The results of test case 16 show one of the mentioned problems. Although the test application is currently not visible, since the screen locking is active, the user is able, by pressing the *Home*-button, to change the state of the current application to *running*. Following the official documentation the application should be visible, while it is running. In test case 18 the test application was started externally by using a USB-cable and *Android Debug Bridge (ADB)*. ADB is one of the official Android development tools, with which developers, for instance, can remotely install, start and remove applications on a device. In the test cases 19 – 21 the configuration of the device (e.g. vertical or horizontal orientation) is changed, while the test application is in state *running* (test case 19), *paused* (test case 20) and *stopped* (test case 21). Test case 20 leads to an error, if executed in the Android emulator. If screen locking is active (activity is in state *paused*) and the developer changes the orientation of the emulator, the application is restarted, as listed in Table I, and remains in state *running* instead of *paused*, being in the foreground visible to the user without screen locking. On a real device this test is executed properly, so that the test application results again in the *paused*-state, after configuration change. In test case 23 the alarm clock was set before the test application was entered. Test case 25 was executed in the Android emulator, by simulating a low battery status via telnet-commands to the emulator. This test cannot be executed on a real device, while the log information is being watched, since connecting the device to a computer leads to charging the battery of the device. So the device battery will not be discharged, while the device is connected to a computer.

Test case 26 shows an interesting behavior of the main test activity in combination with a sub-activity. After the sub-activity, which only partially overlays the main activity, is started, the state of the main activity changes to *paused* by `onPause()`. Pressing the *Home*-button the main activity is stopped (`onStop()`). Returning again to the test application, the main activity is not set to *running*, but to *paused* (`onRestart()` followed by `onStart()`), since the sub-activity is still running and in the foreground. If the user presses the *Home*-button again, the main activity is stopped `onStop()`. So the main activity was started and

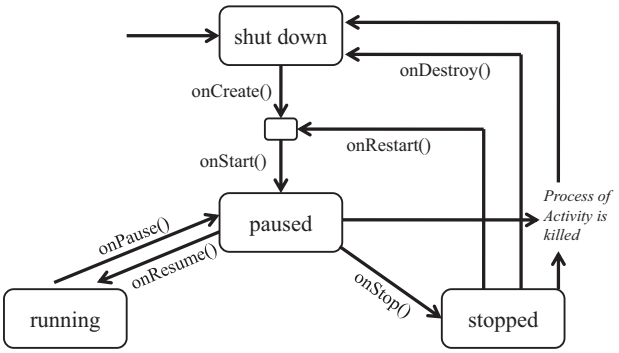

Figure 6. Activity Lifecycle Model

stopped, without reaching the state *running*. The transition sequence of `onStart()`, being followed immediately by `onStop()`, is not possible in the official lifecycle model (see Fig. 2).

Activity Lifecycle Rebuild: Figure 6 shows the rebuilt Android activity lifecycle as a model, derived from the described test results. It consists of four states: *shut down*, *paused*, *running* and *stopped*. The additional state between *shut down* and *paused* has no label and is printed smaller than the other states, since an activity does never remain in this state. An activity enters this state briefly either by `onCreate()` or by `onRestart()` and leaves it immediately by `onStart()`. The transitions are labeled with the lifecycle callback methods also given in the official lifecycle model (see Fig. 2).

The resulting model shows, for instance, that an application in the state *running* cannot be killed. In each test, an application first was paused, before being killed. This means to the developer that data should be stored persistently in the `onPause()`-method, since this is the only method, that is definitely called, before a running application is killed. In the states *paused* and *stopped* an application might get killed by the system, without any callback method being executed. In this case neither data can be stored nor open connections appropriately closed any more.

Another interesting fact, that the resulting model points out, is that after `onRestart()` and `onCreate()` immediately `onStart()` is called. There is no possibility to call `onRestart()` or `onCreate()` without `onStart()` being called immediately afterwards. This means to the developer, that independent of starting or restarting an activity `onStart()` is called each time. So establishing network connections or initializing data should be done in `onStart()` instead of `onCreate()` or `onRestart()`, because else the code is redundant. This might lead to errors if the redundant code is changed in one method, but not in the other.

The rebuilt model also shows what test case 26 pointed out: It is possible that `onStop()` is called immediately after `onStart()`, without running the activity by `onResume()`. For instance, this could lead to an error if the developer closes a connection in `onStop()`, which he establishes in `onResume()`. Since `onResume()` is not called, the connection cannot be closed, which might cause runtime errors. This is one of the most obvious sequences that is not possible in the official activity lifecycle model (see Fig. 2).

B. Reverse Engineering iOS Application Lifecycle

This section describes the reverse engineering of the iOS 4.0 application lifecycle.

Full Implementation of Lifecycle: In an iOS test application we overwrite the following methods, given by the official iOS 4.0 documentation:

- `application:didFinishLaunchingWithOptions:`
- `applicationDidBecomeActive:`
- `applicationWillResignActive:`
- `applicationDidEnterBackground:`
- `applicationWillEnterForeground:`
- `applicationWillTerminate:`

Log Injection: Like Android, iOS also provides a built-in functionality to log information. The `NSLog` function is part of the *Foundation Framework* which is available in the iOS as well as in recent Mac OS. In the tests we log information by the following method call:

```
NSLog(@"App1: application:didFinish
              LaunchingWithOptions:");
```

This example shows the log message in the `application:didFinishLaunchingWithOptions:`-method of the application App1, which is our ID of the main application. Like in our Android tests each application has an unambiguous ID in the log messages and each method is represented by its method name.

Transition Trigger Detection: We execute the iOS test application on an iPhone 4. Table II presents triggers to the test application and the reactions of the application. For reasons of clarity and limited space in this paper we use in the table `didFinishLaunchingWO:` as abbreviation for `applications:didFinishLaunchingWithOptions:` and remove in all other method names the word `application` (e.g. `applicationWillResignActive:` is abbreviated by `WillResignActive:`). For test case 2 the application code had to be changed from test case 1 so that the application executes work in background. We further did introduce a transition label, which is no callback method from iOS: *no more background work*. This is necessary, since depending on the fact, if the application is performing background work or not, different lifecycle methods can be triggered. This is also made clear in the rebuilt model (see next step). In test case 9 data was logged persistently to a file, since the device was not connected to

Table II
TRIGGER CATALOG FOR iOS 4.0 PLATFORM

Trigger	Reaction
(1) start application with intention to work in foreground (by clicking on the application item on the home screen)	`didFinishLaunchingWO:` `DidBecomeActive:`
(2) start application with intention to work in background (by clicking on the application item on the home screen)	`didFinishLaunchingWO:` `DidEnterBackground:`
(3) after (1) receive an incoming call	`WillResignActive:`
(4) after (3) accept the incoming call	`DidEnterBackground:`
(5) after (3) decline the incoming call	`DidBecomeActive:`
(6) after (4) end the current call	`WillEnterForeground:` `DidBecomeActive:`
(7) after (4) if the application has no code to be executed in the background	no more background work
(8) after (4) application is killed due to low resources	`WillTerminate:`
(9) after (1) devices shuts down due to low battery	`DidEnterBackground:` `WillTerminate:`
(10) after (1) press *Home*-button	`WillResignActive:` `DidEnterBackground:`
(11) after (1) shut device down by pressing *On/Off*-button	`WillResignActive:` kill
(12) after (7) application is killed due to low resources	kill
(13) after (7) end the current call	`WillEnterForeground:` `DidBecomeActive:`
(14) after (1) press *Home*-button (only on iOS 3 and older)	`WillResignActive:` `WillTerminate:`

the computer. If it was connected to the computer, it would not discharge. In contrast to the Android emulator, the iOS simulator is not capable of shutting iOS down due to a low battery level. We present test case 14 to show where an important difference was made to the iOS application lifecycle, by introducing multitasking. With iOS 3 and older versions an application was terminated, each time the user pressed the *Home*-button. Further test cases on the iOS did not lead to a different lifecycle behavior. The rebuilt application lifecycle model in the following paragraph shows that with multitasking this is not the case any more.

Application Lifecycle Rebuild: Figure 7 presents the rebuilt application lifecycle of iOS 4.0 applications. For reasons of clarity and limited space in this paper the method names are abbreviated like in the previous step. Like in the Android lifecycle there is no possibility to kill an active application without any method being invoked. Either `applicationWillResignActive:` or `applicationDidEnterBackground:` are called before the application is shut down. Further, the model shows why the *no more background work* transition is needed. From the *suspended* state an application can be killed, without any callback method being invoked. From the *background* state this is not possible.

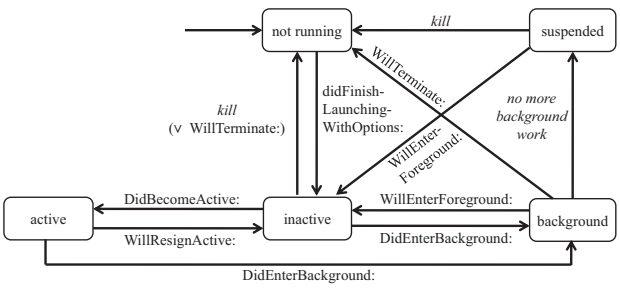

Figure 7. iOS Application Lifecycle Model

C. Reverse Engineering Java ME MIDlet Lifecycle

Concerning reverse engineering of the Java ME MIDlet lifecycle we faced different interesting issues. This section describes the MIDlet reverse engineering process along with the found issues.

Full Implementation of Lifecycle: In order to reverse engineer the MIDlet lifecycle we first create a test MIDlet based on MIDP 2.0 specification, which overwrites the constructor as well as the callback methods `startApp()`, `pauseApp()` and `destroyApp()`. Additionally the test MIDlet displays three buttons: One button triggering `notifyPaused()` on the current MIDlet, another button triggering `notifyDestroyed()` and with the third button the developer can configure, whether a `MIDletStateChangeException` is thrown if `destroyApp(false)` is called.

Log Injection: We test the behavior of the MIDlet lifecycle on several emulators as well as on different real devices. On real devices it is not always possible to print log information in a console. Instead, we use a `Logger`-object of the `net.sf.microlog` (from www.microlog.sourceforge.net) package to display the lifecycle data in an interface form within the test MIDlet GUI and save the logged data persistently using the *Record Management System* of the `javax.microedition.rms` package. For instance, the log entry in the `startApp()`-method in a MIDlet called *MIDlet1* is:

```
logger.info("startApp() called.");
```

Transition Trigger Detection: In our tests we executed the described test MIDlet in emulators and devices with different underlying platforms and mobile operating systems:

- Emulator of the *Java ME SDK 3.0*
- Device *Nokia E71* and emulator of the *Nokia Symbianˆ3 SDK v0.9* each with *Nokia S60*
- Device *Nokia 6300* and emulator of the *Nokia Series 40 5th Edition SDK* each with *Nokia Series 40*
- Device *VPA compact II* with *Windows Mobile*
- Emulator with *Motorola OS*, emulator with *Motorola UIQ* and emulator of two *MOTOMAGX* devices, *MOTOMING A1600* and *MOTO Z6w*, from the *MOTODEV SDK for Java ME v3.0*

The main result of the tests is: Although all underlying platforms support the MIDP 2.0, the behavior of our test MIDlet lifecycle differs significantly between the platforms.

The underlying mobile platforms often do not implement the MIDlet lifecycle model of the MIDP 2.0 exactly. A call of `notifyPaused()` in the *active* state often leads to problems or does not cause the expected behavior. For instance, if `notifyPaused()` is called on the device or the emulator with Nokia S60 or Series 40 platform, the MIDlet is not transferred from *active* into *paused* state. In tests with the emulators from the MOTODEV SDK we found some limitations and bugs in all kinds of emulators. For example, after a call of `notifyPaused()` the GUI of the test MIDlet vanishes from screen, as specified by MIDP 2.0. But if the user opens the MIDlet again, the constructor and afterwards `destroyApp(true)` is called. Hence, `startApp()` is not called according to the specification and the MIDlet does not get back to the foreground. The Windows Mobile device, on the other hand, matches almost all specified lifecycle elements. Only if `notifyPaused()` is called on the Windows Mobile device, the MIDlet shows a strange behavior. It freezes, `pauseApp()` and `startApp()` are called, and the MIDlet does not react on any user input, despite pressing the button which closes the MIDlet. On all tested Nokia S60 and Series 40 devices and emulators `pauseApp()` is never called by default. So when an incoming call occurs, the active test MIDlet on the Windows Mobile device changes its state to *paused*, since the AMS calls its `pauseApp()` method. But on Nokia S60, respectively Nokia Series 40 platform, in this case no callback method is called by default. Therefore, contrary to the test concerning the android activity and iOS application lifecycle, no general statements about triggers of the lifecycle callback methods are possible. And thus we cannot give a set of test cases that trigger every possible transition sequence in the MIDlet lifecycle model.

MIDlet Lifecycle Rebuild: Due to the different behavior of test MIDlet on divers underlying platforms it is not possible to rebuild the MIDlet lifecycle unambiguously. If a developer is interested in the MIDlet lifecycle on a specific device, he can apply the four steps explained in Section III and reverse engineer the application lifecycle for this device. But our results show that such results are mostly not applicable to other devices. Thus, the MIDlet lifecycle needs to be reverse engineered for each Java ME device.

V. Conclusion

For high quality mobile applications the correct implementation of the application lifecycle is crucial. This is especially true when mobile devices are not used as pure end user devices – where occasional errors might be tolerable – but instead are used as control interfaces for industrial applications, e.g. as on-board controllers in vehicles like busses, trams, rail or road trains. Here malfunctioning must be avoided as it might imply high cost for the user and penalty payments for the software provider.

We have shown in this paper that lifecycles of contemporary (iOS and Android) as well as older mobile platforms (Java ME) have issues with the official lifecycle models and corresponding documentation. We present a way to reverse engineer any mobile application lifecycle in four steps. Further, we prove the applicability of the introduced method to the mobile platforms iOS, Android and Java ME. Thus, we find for each of the three platforms either errors in the official models, inconsistencies in the documentation or lacks of information in both. The three case studies also highlight, where problems with different platforms and corresponding development tools (e.g. simulators and emulators) might arise and we present some solutions, how to deal with them. To facilitate reverse engineering of other mobile platforms we provide two catalogs with various triggers for lifecycle callback methods. As future work we plan to evaluate the applicability of the presented method to further current mobile platforms and to check its adaptability to platform changes (e.g. in Android 3.0).

Acknowledgment

This work was supported by the UMIC Research Centre, RWTH Aachen University Germany.

References

[1] R. Rischpater, *Beginning Java ME Platform*. Berkeley, CA, USA: Apress, 2008.

[2] T. Systä, "Understanding the Behavior of Java Programs," in *Proceedings of the 7th Working Conference on Reverse Engineering*, 2000, pp. 214 – 223.

[3] E. Stroulia and T. Systä, "Dynamic Analysis for Reverse Engineering and Program Understanding," *ACM SIGAPP Applied Computing Review*, vol. 10, pp. 8–17, April 2002.

[4] M. Shevertalov and S. Mancoridis, "A Reverse Engineering Tool for Extracting Protocols of Networked Applications," in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 229 –238.

[5] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Transactions on Software Engineering*, vol. 32, 2006.

[6] A. Al-Gahmi, C. John, J. Cook, and B. Du, "Supporting Quick and Dirty CORBA Introspection and Manipulation," in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003, pp. 228 – 237.

[7] R. Meier, *Professional Android 2 Application Development*. Indianapolis, IN, USA: John Wiley & Sons, 2010.

[8] R. B. Hayun, *Java ME on Symbian OS: Inside the Smartphone Model*. West Sussex, England: John Wiley & Sons, 2009.

[9] G. J. Myers, *The Art of Software Testing*, 2nd ed. Hoboken, NJ, USA: John Wiley & Sons, 2004.