AndroLIFT: A Tool for Android Application Life Cycles

Dominik Franke*, Tobias Royé†, and Stefan Kowalewski‡
Embedded Software Laboratory
Ahornstraße 55, 52074 Aachen, Germany
{*franke, †roye, ‡kowalewski}@embedded.rwth-aachen.de

***Abstract*—The states and state transitions of mobile applications - often referred to as application life cycle - play a crucial role in high quality applications. An incorrect life cycle implementation might lead to unexpected application behavior and data loss. However, yet there are no tools available for supporting developers to implement the application life cycle correctly and to test application life cycle-related properties. This work presents the integrated tool AndroLIFT, consisting of two parts, for supporting the correct implementation of Android application life cycles. One part supports implementing and allows monitoring of application life cycles, even of multiple applications being in different states. The second part implements a unit-based testing approach, providing the possibility to test life cycle-related properties. AndroLIFT is implemented as an Eclipse plug-in to be integrated with the Android Developer Tools.**

***Keywords*-*application life cycle*; *unit-based testing*; *development tools*; *software quality*; *Android*.**

I. Introduction

Application life cycles describe the different process related states and state transitions of an application. Figure 1 presents the life cycle of an Android 2.2 Activity. An *Activity* is an Android application, which has a graphical user interface (unlike services). An Activity can be in one of four states:

- It is *shut down*, if it was not started, yet, or if it has been destroyed. The Activity holds no data in RAM.
- An Activity is *stopped*, if it is not visible to the user, e.g., another Activity currently holds the user focus.
- In the state *paused* the application might still be visible to the user, but it does not hold the user focus, e.g., an *incoming call*-dialog covers a part of the user interface.
- If an Activity is *running*, it usually is in foreground and holds the user focus. We show in [1] that this is not always the case. On Android, at each moment in time only one Activity can be in this state.

The two states s_1 and s_2 are intermediate states, in which the application never remains for a long period of time. The transitions are labeled with various method names, e.g., `onCreate()` and `onStart()`. These methods are callback methods, triggered by the Android system in case of a state change. But, not all state changes cause the execution of callback methods. The transitions labeled with *kill* mark state changes, in which the application is killed by the Android system, without invocation of any callback methods. Reasons for such killing might be lacking resources or an application crash. In this cases, the application has no possibility to react on a state change. But, to react on regular state changes, the developer can override the corresponding callback methods.

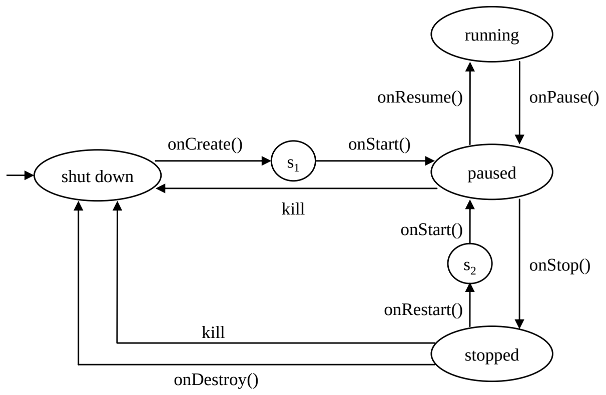


Figure 1. Android Activity Life Cycle [1]

Due to restricted resources and limited input/output capabilities, usually, modern mobile platforms, like Android, iOS and WP 7, have only one active user interface application running (plus some background services). This policy requires a special kind of scheduling (see Fig. 2). Each time a new GUI-application shall be opened, the currently running application first has to be stopped. For instance, on Android the running application first changes its state from *running* to *paused*. Then, the new application changes its state from *shut down* to *paused* and then *running*. Next, the other paused application is stopped and remains in this state. During this application-switch, already multiple life cycle callback methods are called (see Fig. 2). In each of the callback methods the application might have to turn off/on connections, hardware modules (e.g., Bluetooth, Wi-Fi, GPS, ...) or store data. An incorrect or insufficient implementation of the application life cycle might lead to unexpected application behavior and thus to bad user experience, poor usability and data loss [2], [3]. For instance, we pretend that *application A* in Fig. 2 makes use of the GPS module. It releases the module in `onStop()`, since the developer assumes that *application A* is first stopped,

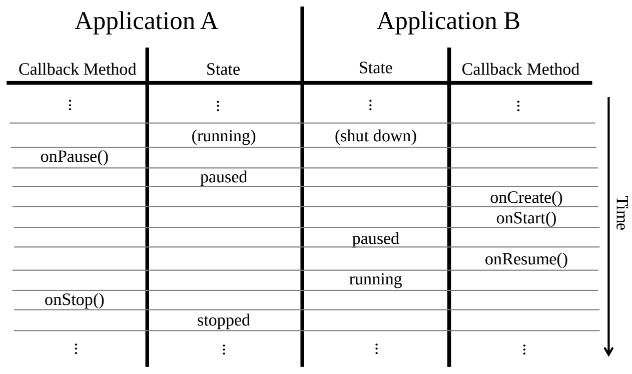


Figure 2. Scheduling two Applications on Android [4]

before *application B* is started. But, Fig. 2, which is the true scheduling on Android 2.2 [4], shows that *application B* is running before *application A* is stopped. If *application B* wants to access the GPS module when starting, it will fail, since *application A* still uses the module. After *application A* releases the module in `onStop()`, no life cycle callback methods of *application B* are called. The GPS module remains unused and *application B* does not use it, except if *application B* would actively poll for it, which is no life cycle action.

There are no tools available, yet, to help developers to find this certain kind of errors, implement the application life cycle or test life cycle-related properties of mobile applications. In this paper, we present *AndroLIFT*, a tool which helps developers of mobile applications to analyze, implement and test application life cycles. AndroLIFT is implemented for Android applications, but the concepts behind it can be implemented for other mobile platforms, too. We chose Android for a first implementation, since it is currently the most widespread mobile platform. Additionally, Android itself and the developer tools for Android are available open source. This allows to integrate AndroLIFT as a plug-in into the available Android Eclipse framework. Neither iOS nor Windows Phone, two competing mobile platforms, are available open source. The first functionality of AndroLIFT we present, supports the implementation of the application life cycle by allowing to monitor application life cycles during runtime in a life cycle view. It also eases the implementation of the life cycle, e.g., overriding the life cycle callback methods, by connecting a graphical view of the application life cycle to the source code editor. The second functionality integrates the life cycle testing approach from [4] into the Android Eclipse plug-in. It provides a user-friendly graphical interface to the life cycle testing library, eases implementation of the test approach, and connects it directly to the life cycle view.

The paper is structured as follows: Section II presents details about some Android developer tools, of which AndroLIFT makes use of. In Section III, the life cycle view is introduced. This view is extended by the testing functionality in Section IV. Section V concludes this work.

II. Background

This Section introduces the tools on which the AndroLIFT library and plug-in are based on. First, the Android Logcat tool, a logging tool for Android devices, as part of the Android SDK, is presented. Second, a brief description of the Android Development Tools, for development of Android applications with Eclipse, is given.

A. Android SDK, ADB and Logcat

The *Android Software Development Kit (SDK)* is a bundle of various tools, applications and documentation to develop software for the Android platform [5]. Since the Android SDK is a very rich bundle, but we only need few of those components for this work, we do not explain too much about the Android SDK itself. Therefore, we refer to the official Android SDK references.

One of the core components of the Android SDK for application development is the *Android Debug Bridge (ADB)*. ADB is a command line tool, which allows to communicate between a development machine and an Android device or emulator. For instance, it provides the possibility to send data to a device, remotely install and remove applications and forward ports. It also allows to receive log information from the device using the *Logcat*-tool. Android's Logcat allows on a development machine to view debug output from an emulator or connected device. It is the main logging mechanism on Android. Next to log messages sent by applications using the *android.util.Log*-class, it also provides various system information, as stack traces in case of an error and kill information, if a process is killed.

Logcat has a structured way of logging. For instance, each log information can be attached with certain information. Printing a log information with priority *debug* looks from the perspective of the developer as follows:

```
Log.d("MyTestClass",
      "Connection to server failed.");
```

The corresponding Logcat-output looks like:

```
D/MyTestClass( 1633):
       Connection to server failed.
```

We use this Logcat tool to send information about the life cycle state of an application to AndroLIFT.

B. Android Development Tools

The *Android Development Tools (ADT)* is a tool collection for development of Android applications with the Eclipse IDE. The ADT Eclipse plug-in extends Eclipse with different features, like Android projects, building and debugging Android applications, SDK tools integration, Android XML editor and integrated Android framework documentation [6].

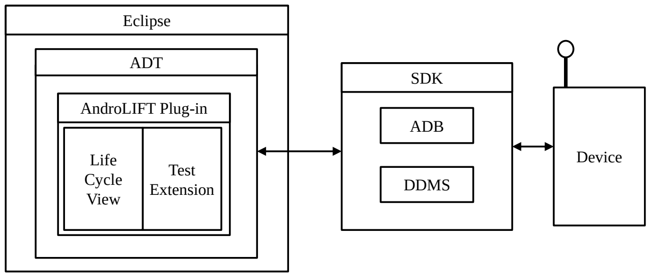


Figure 3. Integration of AndroLIFT into ADT

It is the common way to develop Android applications with ADT and Eclipse.

As these tools are available open source, we build our AndroLIFT plug-in on top of ADT. The advantage is that Android developers used to Android development with ADT get an integrated approach in a well-known environment. The integration of AndroLIFT into Eclipse and ADT is sketched in Fig. 3. AndroLIFT is fully integrated into the ADT environment. It consists of two parts, introduced in Sec. III and IV. ADT uses the Android SDK tools to communicate with devices. For the following work, especially the SDK-tools ADB and *Dalvik Debug Monitor Server (DDMS)* are important. DDMS allows further debugging services, like radio state information, screen capture on device and incoming call spoofing [7]. It usually connects to ADB to provide its full functionality. So, following the architecture in Fig. 3, the AndroLIFT plug-in never communicates directly with the device, but uses the same communication channels as ADT does: over ADB and DDMS.

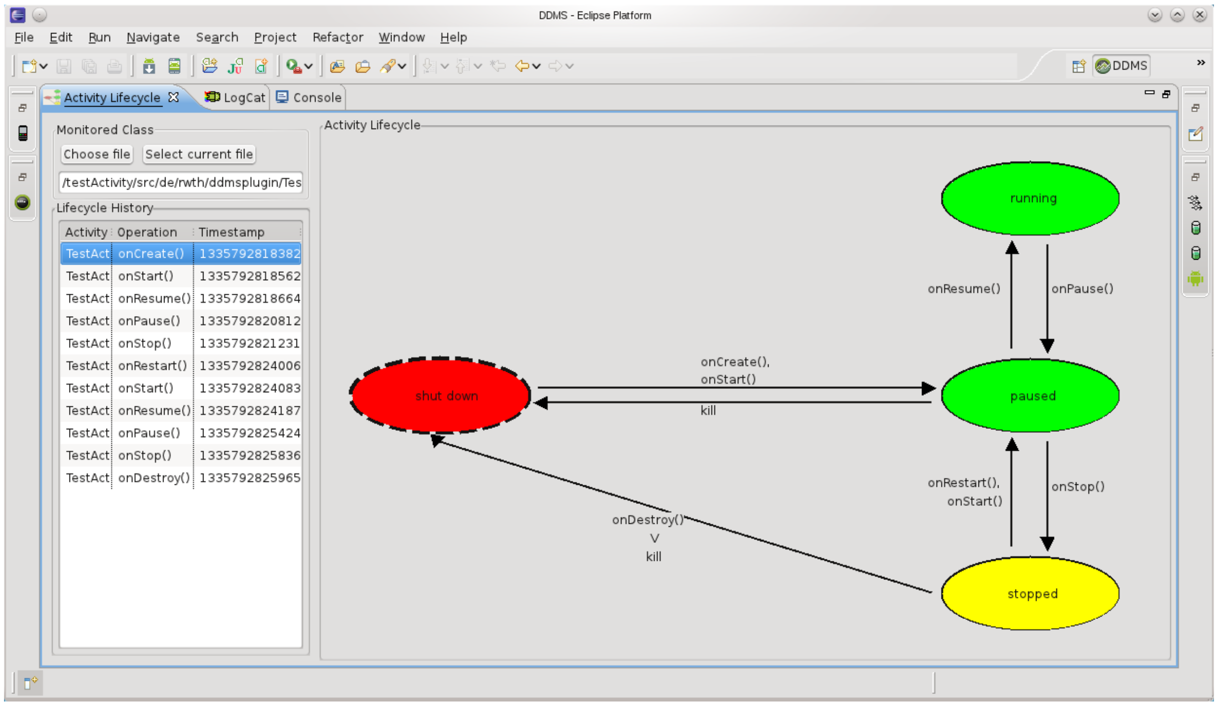
III. Life Cycle View

To find out in which life cycle state an application is, the developer has to override the corresponding life cycle callback methods and print the corresponding information. There exists no other way to get this information on all mobile platforms like Android and iOS. But, as mentioned above, the life cycle of a mobile application is an important component for high quality applications. A graphical representation of the life cycle would help to examine the life cycle, its behavior during state changes and corresponding callback methods to react on certain events. We present such a view for the Android platform as part of ADT.

Figure 4 shows a screenshot of the AndroLIFT life cycle view of an Android Activity. The life cycle is taken from a reverse engineering approach [1], which is shown in Fig. 1. The intermediate states are left out (see Fig. 1), since for the developer it is only important what the resulting callback sequence on the corresponding transition path is, e.g., `onCreate()` is followed by `onStart()`. As a first step the developer has to specify on the left side of the life cycle view, which Activity of which package shall be monitored. Therefore the Activity can be executed on the Android emulator or a USB-connected real device. Then the current state of this Activity is marked by a dashed line. For instance, the Activity in Fig. 4 is currently in the state *shut down*. Next to the different states of the Activity, all available callback methods and kill-transitions are displayed as labels of the state transitions. If the monitored Activity changes its state, e.g., from *shut down* over *paused* to *running*, the corresponding path and states in the view are animated. To make the state changes comprehensible and traceable for the developers, the animations do not run in real-time, but slightly delayed. First, the transition labeled `onCreate(), onStart()` would be marked, followed by the state *paused*, transition labeled `onResume()` and finally state *running*, each marked for 500ms. Additionally, each call of a callback method is logged in the life cycle callback history view, presented on the left side in Fig. 4. It prints the name of the Activitiy (important in case of multiple running Activities), name of the callback method executed in that Activity and a timestamp of the call, to understand the order of the callback methods. With such a view on the application life cycle, the developer easily can find out the different state changes as a consequence of a certain event, e.g., incoming call or SMS.

If the developer knows and sees how his application behaves during runtime regarding its life cycle and which callback methods are called, he can easily implement a correct life cycle behavior. To ease the implementation of the life cycle, we added another feature to the life cycle view. By right-clicking a callback method, a menu pops up, with which the developer immediately can jump to the corresponding callback method in the source code editor. An example is given in Fig. 5, where the developer is about to modify the life cycle callback method `onStart()`. Additionally, if the corresponding callback method is not overridden, yet, AndroLIFT automatically overrides the callback method and places the cursor to the correct position in the source code editor. The developer immediately can start implementing the life cycle behavior.

Since the Android system itself, as all other modern mobile platforms, do not give any information about the current state, the application under test has to do so. It has to report the state of its life cycle to the life cycle view, each time the state changes. This information is needed by the life cycle view to trace the life cycle during runtime. This can be done in two different ways. One way is to extend an Activity class, called *DebugActivity*, which AndroLIFT provides. This class has already all code, which is needed by the life cycle view, encapsulated. This includes the initialization of the connection between the AndroLIFT plug-in and the application under test. It also automatically forwards state information to AndroLIFT via Logcat. If the developer does not want to use the DebugActivity, the code has to be placed manually in the corresponding applications, which is no big effort, either. For the `onPause()`-method the injected code



Activity	Operation	Timestamp
TestAct	onCreate()	1335792818382
TestAct	onStart()	1335792818562
TestAct	onResume()	1335792818664
TestAct	onPause()	1335792820812
TestAct	onStop()	1335792821231
TestAct	onRestart()	1335792824006
TestAct	onStart()	1335792824083
TestAct	onResume()	1335792824187
TestAct	onPause()	1335792825424
TestAct	onStop()	1335792825836
TestAct	onDestroy()	1335792825965

Figure 4. The Life Cycle View allows to monitor the Life Cycle nearly in Real-time

looks as follows:

```
Log.d(this.getClass.getName(),
     "onPause() called.");
```

The corresponding callback to the super class is mandatory in callback methods on Android and for AndroLIFT certain information containing the names of the package, Activity and life cycle callback method have to be printed to the Logcat tool. This has to be done for each life cycle callback method. This information is fetched by the life cycle view and processed accordingly. For instance, by knowing, which life cycle callback method was recently called, AndroLIFT knows the current state of the application.

IV. Life Cycle Testing

In a previous work [4], we present an approach to test life cycle-related properties on mobile platforms. This unit-based testing approach sees one Activity as a unit, regarding its life cycle. Life cycle state changes on modern mobile platforms are usually only triggered by the underlying system, not directly by the application itself or by another application [8]. For instance, if an application requests to be paused, the underlying system decides if and when to pause the application. In this sense a mobile application is separated regarding its life cycle, and thus in this sense a unit. This is not unit-testing in the original sense, where smallest testable part is separated and tested [9]. On Android, we see one Activity as a unit. We trigger the environment, e.g., making an incoming call to the Android system, and observe the reactions of the Activity regarding its life cycle.

To specify the expected behavior of an application we use assertions. An assertion can be derived from a part of the specification of the application, e.g., *if the user receives an incoming call while writing an e-mail, do not loose the already composed e-mail text*. On Android, if the user is writing an e-mail, the corresponding application must be in the state *running* (see Fig. 1). We know from [1] that an incoming call causes an Android Activity to be paused. Thus, following our testing approach, in `onPause()` assertions need to be defined, which store the current content of the affected text fields (subject, main text, etc.) and a reference to the text fields. The object reference is needed to be able to check those text fields after resuming the application. After the call, the application resumes again, which means that the callback method `onResume()` is invoked (see Fig. 1). So the previously defined assertions have to be checked in `onResume()`. The stored text is compared to the current text in the affected text fields and the test results are printed to the user. For more detailed information on testing life cycles of mobile applications, we refer to [4].

We implemented this approach for Android as a library, called *AndroLIFT runtime assertion library*. The package structure of this library is sketched in Fig. 6. Due to reasons of clarity, not all classes are displayed in this figure. The *LCAssertions*-class is the main class of the library. It handles, stores and checks all assertions. The *Util*-package

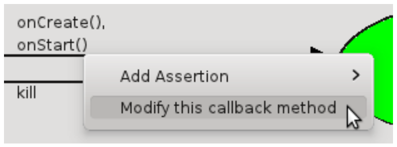


Figure 5. The Life Cycle View assists in implementing Life Cycle Behavior

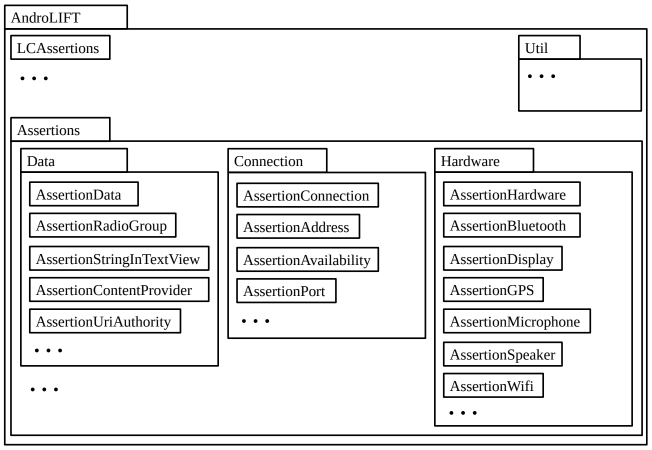


Figure 6. Package Overview of the AndroLIFT Runtime Assertion Library

holds different utilities, like database schemas for storing assertions. The *Assertions*-package contains various types of assertions:

Data Assertions: All assertions regarding data persistence, e.g., content of text fields, choice of radio buttons and list selection.

Connection Assertions: Assertions for checking if some kind of connection, e.g., IP/TCP or Bluetooth, is still available and active.

Hardware Assertions: Assertions about the status of hardware components, e.g., checking if GPS or Bluetooth is on.

The developer uses the different types of assertions to specify requirements to life cycle-related properties of his application and passes them to *LCAssertions*. This core class of the library manages the definitions and checks of the assertions depending on the current application state. By knowing the current state, AndroLIFT is able to check all assertions in the corresponding states. Since mobile applications might be killed, e.g., due to lacking resources, the library is also able to store assertions persistently. Therefore it uses the storage possibilities of the application under test, like the database on Android. With this concept, the developer can also check requirements like:

The application might not loose the content of an e-mail, even if it is killed by the system due to low battery.

The corresponding assertions are then stored in the database, which is a persistent storage. After the corresponding test case is executed, e.g., low battery is simulated with the Android emulator, and the application is returned, the assertions can be restored from the database and checked.

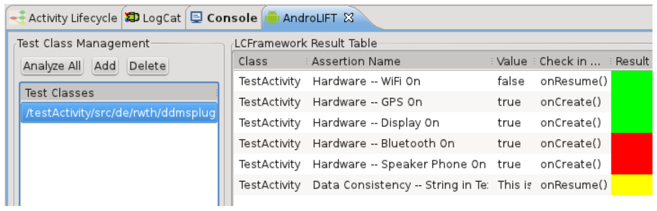
The following code presents an example usage of the AndroLIFT library:

```
@Override
public void onPause(){
  super.onPause();
  Assertion a = Factory.
    createDataAssertion(STRING_IN_VIEW,
      textView1);
  androLift.assertThat(ON_RESTART, a);
  androLift.onPause();
}
```

With the help of an assertions-factory, the developer defines an assertion *a* about a string value in a text view object *textView1*. The library fetches the current text from the text view and stores it automatically. Further, the developer specifies that *a* shall be checked in the callback method `onRestart()`. The last line in this example tells AndroLIFT that `onPause()` is called, so the state changes to *paused*. Due to the restricted possibilities to get information about the current application state on Android, AndroLIFT needs to get this information from the application under test. So just like with the life cycle view (see Sec. III), the application under test needs to make a call to AndroLIFT in each life cycle callback method. Regarding the example above, for the callback method `onPause()` it is the line `androLift.onPause()`.

On top of this life cycle testing API, we developed a graphical user interface, which we integrated as an extension to the life cycle view plug-in. In this case there are various advantages of a graphical user interface over a library: The library itself was not integrated into the well-known ADT. Since the life cycle view is integrated into these tools, the testing extension is. The usability of the testing library is enhanced by this integrated solution. Further it is easier to learn and more intuitive to use than on code-level with corresponding code-level documentation.

With the graphical test extension, the user can create an assertion by right-clicking the corresponding life cycle callback method, in which the assertion shall be checked. With only few clicks the user is able to create the same assertion as given in the code above. First, he has to decide, which type of assertion he wants to define. Second, he needs to specify, where the object, e.g., a text view, is defined, since on Android user interface objects can be defined in Java as well as in XML. Finally, the developer needs to define the method, in which the assertions shall be defined. It will be checked in the method he right-clicked before. Figure 7 shows the output-view of the AndroLIFT test-extension. On the left side the developer can choose, which



Class	Assertion Name	Value	Check in ...	Result
TestActivity	Hardware -- WiFi On	false	onResume()	🟩
TestActivity	Hardware -- GPS On	true	onCreate()	🟩
TestActivity	Hardware -- Display On	true	onCreate()	🟩
TestActivity	Hardware -- Bluetooth On	true	onCreate()	🟥
TestActivity	Hardware -- Speaker Phone On	true	onCreate()	🟥
TestActivity	Data Consistency -- String in Te:	This is	onResume()	🟨

Figure 7. Output of the AndroLIFT Plug-in

test results from which Activity he wants to see. If his application contains multiple Activities, they are listed on the left side. After clicking on one of the listed Activities, the corresponding test results are printed in the table. The developer can see the type of the assertion, the attached value as well as the result. If the assertion passed, the result field is filled green. If an assertion did not pass, the result field is printed red. If the assertion has not yet been checked, e.g., since only the defining but not yet the checking callback method has been executed, the result field prints yellow. This way of reading test results is far more user friendly and less error prone than checking test results in a log file, as with the pure AndroLIFT runtime assertion library.

V. Conclusion

Application life cycles play an important role in the area of mobile applications. An incorrect or insufficient implementation of the life cycle might cause unexpected behavior of the application, leading to bad usability and even data loss. Until now, there are no tools available to analyze and test application life cycles.

In this paper, we presented AndroLIFT, a tool which helps the developer to monitor the life cycle, assists him in implementing it and testing life cycle-related properties. AndroLIFT is written as an extension to the ADT, the common way of developing Android applications with the Eclipse IDE. With the life cycle view the developer can observe and analyze the life cycle of his Android application. Developers easily learn about the behavior of the application life cycle to certain triggers, like an incoming call, and with which callback methods one can react appropriately. Further, with right-clicking the corresponding life cycle callback methods in the life cycle view he can quickly implement assisted the life cycle of his application. With the test extension of the AndroLIFT plug-in the developer has a user-friendly way to use the AndroLIFT testing library. From within the life cycle view he can create life cycle assertions, using the corresponding GUI. During and after test execution, e.g., simulation of an incoming call, the test extension of the life cycle view presents the results in a well-readable way, aligned to the well-known JUnit-testing tools.

By helping to learn and understand the life cycle of Android applications quicker and better, developers get a good feeling for the behavior of the life cycle to certain events. They can immediately see, with which life cycle callback methods they can react appropriately to certain life cycle triggers. With the test extension the developer can specify test cases by creating assertions in an intuitive and user-friendly way. The tool handles automatically all source code injections (except a few), which are necessary for working with the AndroLIFT runtime assertion library. Additionally, the life cycle test results are presented in a human readable and comprehensible way.

Acknowledgment

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

References

[1] D. Franke, C. Elsemann, and S. Weise, Carsten Kowalewski, “Reverse engineering of mobile application lifecycles,” in *18th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2011, pp. 283 – 292.

[2] D. Franke, S. Kowalewski, and C. Weise, “A mobile software quality model,” in *12th International Conference on Quality Software (QSIC)*. IEEE Computer Society, 2012, pp. 1 – 4.

[3] D. Franke and C. Weise, “Providing a software quality framework for testing of mobile applications,” in *4th International Conference on Software Testing Verification and Validation (ICST), Berlin, Germany*. IEEE Computer Society, 2011, pp. 431–434.

[4] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol, “Testing conformance of lifecycle-dependent properties of mobile applications,” in *5th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2012, pp. 241 – 250.

[5] S. Komatineni and D. MacLean, *Pro Android 4*. Apress, 2012, vol. 1.

[6] R. Meier, *Professional Android 2 Application Development*. John Wiley & Sons, 2010, vol. 2.

[7] E. Burnette, *Unlocking Android: A Developer’s Guide*. Manning Publications, 2009, vol. 2.

[8] D. Franke, C. Elsemann, and S. Kowalewski, “Reverse engineering and testing service life cycles of mobile platforms,” in *2nd DEXA Workshop on Information Systems for Situation Awareness and Situation Management (ISSASiM)*. IEEE Computer Society, 2012, pp. 16 – 20.

[9] A. Hunt and D. Thomas, *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003, vol. 1.