A Model-based approach to assist Android Activity Lifecycle Development

Conference Paper · October 2020

DOI: 10.1109/ISMSIT50672.2020.9254687

2 authors:

Timothy Ghanem
Birzeit University
3 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Samer Zein
Birzeit University
27 PUBLICATIONS **226** CITATIONS
SEE PROFILE

A Model-based approach to assist Android Activity Lifecycle Development

Timothy Ghanem
Master in Software Engineering
Birzeit University
Birzeit, Palestine
timothy.ghanim@gmail.com

Samer Zein
Department of Computer Science
Birzeit University
Birzeit, Palestine
szain@birzeit.edu

***Abstract*— In Android app development, conforming to the activity lifecycle model is imperative to maintain app robustness and reliability as well as avoiding many issues tied to lifecycle state transitions, such as memory leaks, data preservation, and app crashes. Previous studies have shown that Android developers possess limited understanding and awareness of the activity lifecycle model and the current state-of-the-art Android app development tools and methods provide developers with little support during activity lifecycle development. In this study, we present an approach and a framework that provides a dynamic visual view for the activity lifecycle state transitions during implementation. The approach follows model-based development utilizing DSVL (Domain Specific Visual Language) and is implemented as a proof-of-concept Android Studio plugin. We evaluated our approach through experimentation by real Android developers. Initial results show that our approach can be useful and effective in assisting Android developers.**

Keywords—model-based, model-driven, MDD, android lifecycle, android, android development, lifecycle development.

I. INTRODUCTION

Two primary distinguishing features of Android over any other mobile development platform are that it is: (1) open source and hence free to download and use and (2) built using the Java programming language; one of the most popular and globally used programming languages. The fact that Android is open source has made it the primary choice for smartphone manufacturers and as a result has been adopted in a wide variety of smartphone devices. As a result, it became the primary focus for scientific research targeting the development of mobile applications, which has contributed in the identification of design issues and development of tools to assist developers who build applications that target the Android platform [1].

Proper handling of Android Activity lifecycle state transitions is implemented in the Activity lifecycle callback methods. This is a very important aspect of any Android app development activity in order to avoid app crashes, consumption of valuable system resources. It has been observed that a large spread of memory leaks in the wild are tied to improper implementation of Activity lifecycle callback methods [4], and that the objects that most frequently contain memory leaks in Android apps were the Activity classes [5]. In addition, it has been observed that Android developers lack the necessary knowledge and awareness of the Android lifecycle model [6]. Activity lifecycle implementation failures could be triggered as easily as a change in the orientation of the device. It has been observed that a change in orientation of the device after performing some interactions with Dropbox version 27.1.2 caused it to suddenly stop working [7].

To the best of our knowledge, none of the current state-of-the-art frameworks and tools provides multi-view support to assist Android developers during activity lifecycle implementation.

To that end and in order to aid Android app developers in implementing a better handling of the Activity lifecycle callback methods, we present in this paper a model-driven dynamic multi-view approach, implemented as an Android Studio plugin which presents the developer with a view that reflects the Activity lifecycle callback methods implemented in each Activity class. The approach utilizes two domain-specific languages: textual (DSTL) and visual (DSVL). The view presented by our approach is an additional view to the default textual code view that the developer has.

More formally, this research aims at addressing the following research questions:

1. RQ1: How can model-based solutions be applied to assist Android app lifecycle development?
2. RQ2: To what extent can a multi-view solution affect Android developers?

In addition to the main contribution of this paper. The presented approach contains the following contributions as well:

1. A Domain-Specific Textual Language (DSTL) that can be used to represent an implementation of an Android Activity lifecycle.
2. A Domain-Specific Visual Language (DSVL) to represent, in visual terms, the Android Activity lifecycle.
3. A tool, implemented as Android Studio plugin, that implements an Android Activity class source code to DSTL code that conforms to the presented DSTL, a DSTL to DSVL converter that converts an instance of the DSTL code to another instance that conforms to the DSVL presented. The tool contains a rendering engine that is able to represent any instance of the DSVL.
4. Another contribution is that the implemented tool is designed with extensibility in mind such that it can be easily extended to allow for different customizations.

The rest of this paper is structured as follows: Section 2 presents the motivation behind this research; Section 3 addresses necessary technical background to understand the technical side of proper implementation of the Android Activity lifecycle callback methods; Section 4 discusses a survey of the current state-of-the-art approaches and tools that are used to build Android apps and tools that test and analyze an app lifecycle implementation. Section 5 discusses the proposed research methodology that was used to answer the previous research questions. Section 6 goes over the technical implementation of the presented approach as an Android Studio plugin. Finally, Section 7 discusses the evaluation of the presented approach and the results that represent Android developers satisfaction with the presented approach.

II. MOTIVATION

A user can perform many activities with a smartphone such as switching between apps, change device orientation…etc. As a result of these activities, mobile apps go through different state transitions. In Android, the affected component in the mobile app is called the Activity. An Activity is the main component with which an Android user interacts [2]. Typically, the operating system notifies the app's Activity of all those state transitions in order to allow app developers to maintain an acceptable user experience throughout the app lifecycle. Activity's lifecycle state transitions are typically implemented by using callback methods that are implemented inside the Activity class. Those callback methods are called by the underlying operating system whenever the Activity's state changes [2]. As a result, Android apps are categorized as event-driven applications [22]. A mobile app lifecycle is significantly different from that of a standard Web or Desktop application [26]. For instance, and in the Android case, when a user switches between apps, the foreground app Activity is transitioned to the paused state and the switched-to Activity is transitioned to the resumed state [3]. On the other hand, a desktop app remains a foreground process except that its windows may be unallocated from the screen and a Web application does not even have the paused state.

To analyze Android lifecycle-related issues, there exists a large body of research that presents several categories of approaches such as security analysis [27, 28, 16, 17, 18], black-box testing [7, 4], lifecycle conformance checking [24]...etc. Such approaches attempt to model the app's implementation of the Activity lifecycle in each Activity class in the app and then perform different types of analysis to verify different aspects in the Activity lifecycle implementation. Examples on these aspects are: making sure that no sensitive data is leaked from the app [21]; checking if an app correctly releases the resources it acquired when it is no longer in the foreground; and if the app doesn't purposefully alter its lifecycle for more than what the user expectations are [18].

Model-driven development is yet another approach for mobile app construction from different types of models. In general, these approaches provide an initial boost in development time and hence developer's productivity. Several types of model-driven development approaches are presented in the literature such as visual-driven [9, 10, 12], text-driven approaches [11], or a combination of both [8]. Visual-driven approaches present the developer - or app modeler - with a high-level view where the pages, flow, data are all represented at a high-level abstraction. The model typically conforms to certain type of a domain-specific visual language (DSVL). Then, through a series of model transformations, the final source code for the mobile app is produced. Similarly, text-driven approaches start the modeling process using a high-level domain-specific textual language (DSTL) where the developer (or modeler) describes the mobile app using a textual code that conforms to the domain-specific language. The code is then transformed into the final app source code. Model-driven development could also contain a combination of both the high-level models (i.e. visual (DSVL) and textual (DSTL)) where each model type allows specifying different aspects of the end mobile app.

None of the existing approaches and tools presented in the literature has a dedicated support for developers to write a robust and reliable Android Activity lifecycle handling.

III. BACKGROUND

Android applications are event-driven applications. This means that the app receives event notifications about changes to its state from the underlying operating system. The Activity class in Android is the primary thing with which the user interacts [2]. Each Activity class has its own lifecycle that is implemented by overriding and implementing different callback methods. Activities in Android are managed using stacks. When a new Activity is started, it is placed on the top of the stack and becomes the running activity. The previous activity remains below it and will not come to the foreground until the activity on the top of the stack exits.

Each Activity instance has three key loops in its lifecycle:

1. The **Entire Lifetime** of the activity which starts with the *onCreate* callback through to a single final call to the *onDestroy* callback. After the *onDestroy* callback is called, the Activity is destroyed from the system.
2. The **Visible Lifetime** which starts with a call to the *onStart* callback until a corresponding call to the *onStop* callback. When an Activity is stopped, it is no longer visible to the user. If the user navigates back to the Activity, the *onStart* method is called again.
3. The **Foreground Lifetime** which starts with a call to the *onResume* callback until a corresponding call to the *onPause* callback. When the Activity is paused, it is still visible to the user but not under focus. If the user returns to the Activity, the *onResume* callback is called again. During the paused state, the system may choose to reclaim the memory allocated to the Activity process, and therefore the app process is killed. If the user navigates back to the Activity, the *onCreate* callback is called again.

The Android Developer Guide [2] provides several guidelines on the handling of the Activity lifecycle callback methods with respect to resource allocation and releasing. It is recommended that all "global" state is set up in the *onCreate* method and released in the *onDestroy* method. During the paused state, while the Activity is still visible to the user, it is recommended to maintain the resources that are necessary to show what is needed to the user. In addition, it is highly recommended that any persistent data is written to the storage in the *onPause* method because it is the only method where the Activity is notified to lose the foreground state which is not killable by the system; the *onStop* callback is marked as killable meaning that the Activity may be killed at *any time* without another line of its code being executed after the method returns to the process hosting the Activity. In all cases, the system reserves its right to kill the app process at any time under extreme memory pressure.

IV. LITERATURE REVIEW

A. Model-based Android App Development

Model-driven development or model-driven engineering is a category of software development where the app developer/modeler uses a high-level abstract model to describe the app. The high-level model allows specifying the structure and behavior of the user interface, the structure of the data collected from each page. The model could allow for more advanced scenarios such as backend service communication. Then, through a series of model transformations, the source code of the final app is produced. The developer can then apply final polishing to the code, compile and deploy it to the target device. The model can be either a visual or a textual model. Another synonym for the model is domain-specific language.

Freitas et al. [9] presented a graphical tool to model the business classes of an Android app and called it JBModel. The classes are modelled as UML classes including their relationships which are then transformed through a model-2-text transformer to POJO classes that target the JustBusiness framework which is then responsible to generate the UI, persistence code, and application resources necessary to compile and run the application. The developer is yet to implement the generated POJO classes manually.

A more advanced approach for model-driven development was presented by Vaupel et al. [10] where an app is modelled using a graphical tool which contains three different views for three different models. The data entities and their relationships are described using a *data model*, data management, access to sensors, use of other apps are described using a *process model*, the definition of the pages is described using a GUI model. At runtime, and to avoid redeployment of the app to reflect changes, another model, called the *provider model*, was introduced which contains instances of the data, process, and GUI models which serve as arguments to customize the behavior of the app at runtime.

An example of using pure textual model-driven development is presented in Thu et al., [11] where a rule-based transformation was presented which takes as input a textual model written in Umple and generates an entire app following the MVC pattern. The transformation rules are written in the Drools Rule language.

Rieger et al., [12] wanted a modeling language that did not require software engineering knowledge (technical complexity) and can be understood by domain experts and can be easily interpreted by code generators (avoid GUI oversimplification). They presented a graphical domain-specific language, called MAML, to make this trade-off, accompanied with model transformers to allow generating native apps from the graphical model.

An instance where multiple views are presented to model-driven developers exist in the work of Barnett et al. [8] where the authors presented a tool, called RAPPT, which leverages the two types of models (visual and textual) as two domain-specific languages. The visual modeling language allows specifying high-level structures such as the number of screens, navigation flow between screens...etc. while the textual modeling language allows providing low-level details such as backend services, data schema, authentication...etc. To achieve this, the authors created a shared internal model which the two modeling languages target.

Model-driven mobile app development was also found to be utilized in the generation of context-aware apps which can adapt to contextual parameters such as the user's computing infrastructure, location...etc. Li et al. [13] presented a meta-model for context-aware activity-oriented applications with a platform-independent domain-specific language, called AocML, to describe the application. The domain-specific language is then converted, using a model-to-text transformation to Java code. The generated Java code targets the platform for activity-oriented context which is a Java-based platform for supporting the development and runtime of activity-oriented context-aware applications.

Paspallis et al. [14] wanted to facilitate the development of context-aware applications by presenting an approach for the design of reusable context plug-ins that can be used to monitor low-level context data and infer higher-level information about the users, their computing infrastructure and interaction. A UML profile was presented to be used as a modeling language. The UML model is converted to XMI which in turn is converted to XML/UML2 as expected by the underlying model transformer (MOFScript). MOFScript is then used to generate the actual source code for the context plug-in.

Yigitbas et al. [15] extended model-driven development to create mobile apps with user interfaces that are self-adaptive to contextual information. The authors presented two domain-specific languages: ContextML and AdaptML. ContextML allows defining context properties and context providers that capture relevant context information. AdaptML allows modeling the UI adaptation rules. Several models are presented, Abstract UI model, Domain Model, Context Model, and Adaptation Model. The Abstract UI and Domain models are used to generate the final UI of the app. The Context Model is used to generate Context Services that monitor context information like accelerometer, GPS, brightness, or noise level. The Adaptation Model which references the Context Model allows defining constraints for triggering adaptation rules which reference the affected UI elements in the Abstract UI model. The Adaptation Model is used to generate an adaptation service which monitors information collected by the context service and adapts the final UI at runtime.

It can be observed from the previous survey of the state-of-art approaches for model-driven development of mobile apps that there's no special attention given for the generation of proper handling of the app lifecycle. In addition, even while these approaches present a significant boost in development time and increased separation of concerns, yet they do not provide the necessary flexibility required by the developers and therefore, for apps with a long lifespan, the model-driven approach can only be used to bootstrap the initial app development and any new features, enhancements, or bug fixes have to be performed manually without the model-driven approach, and where the app lifecycle handling is concerned, the proper implementation of the lifecycle callback methods is again left to the developer and the model-driven tool is useless in that case.

B. Android Lifecycle Handling Analysis

Android lifecycle handling analysis starts by presenting a model of the Android lifecycle implementation. Then, given the Android Package (APK) file or the app's source code, a tool will parse the app's source code, develop an instance of the presented model, then analyze the model instance to detect issues with the app's implementation of the lifecycle callback methods.

1) Security Analysis: A considerable portion of the literature which includes a modeling of the Android app lifecycle is concerned with security analysis. Junaid et al., [5] attempted to construct an Android lifecycle model which captures all the states and transitions implemented in the app. The model is then systematically used to derive lifecycle callback events which are then used by taint analysis techniques to detect malicious app behavior. The analysis of malicious behavior with respect to data flow analysis was extended by Li et al., [16] to investigate the effect of the Android Fragment lifecycle on the Activity lifecycle. The authors introduced a control flow graph model for control flow transfers of Fragments' and Activities' life cycles and used the model to perform information leakage detection to eliminate the false positives in current state-of-the-art information leakage techniques which focus only on the Activity lifecycle. To detect Android apps that execute additional actions to hide their malicious behaviors (i.e. stealthy attacks), Junaid et al. [17] presented a static analysis framework which implements a FSM model, which includes a modeling of the Android lifecycle callbacks, to depict actions and action-sequence based stealthy attacks. Another category of apps that alter their behavior are diehard apps which either directly or indirectly alter their lifecycle to avoid being killed by the operating systems. Shao et al. [18] presented a runtime framework which is based on a system-wide app lifecycle model called Application Lifecycle Graph (ALG). The framework builds the ALG at runtime and uses it to realize fine-grained lifecycle control of apps.

2) Android App Testing: Another purpose for modeling the Android app lifecycle was found to assist in the black-box testing of Android apps. While Riccio et al. [7] presented a black-box testing approach that is able to detect crashes and GUI failures which are tied to the Activity Lifecycle implementation, Amalfitano et al. [19] presented an approach to automatically detect memory leaks that are due to bad programming practices with focus on bad handling of system events that are tied to the Android Activity Lifecycle. While The former attempts to detect lifecycle event sequences that can cause a GUI failure or system crash, the latter uses memory analysis to detect object replication in the app's memory. Another attempt to detect context leakages in Android native apps was presented by Toffalini et al.'s [20] static code analysis tool which identifies static fields that reference contexts either directly or indirectly. Potential leaks are then systematically analyzed w.r.t severity.

3) Other Approaches: Other approaches in the literature exist which build a model of the Android app lifecycle to achieve other purposes. To assist analysis tools developers, Guo et al. [21] presented a tool that constructs a generic callback-related model to support path-sensitive analysis. The model can identify connections between different components, path-sensitive conditions, and the handling of the system-driven fine-grained lifecycle callbacks. On the other hand, Perez et al. [22] presented a program representation that can generate sequences of Android lifecycle callbacks that match the runtime behavior. To detect race conditions in Android apps, Hu et al. [23] presented a static analysis tool that is able to model threads, messages, lifecycle Activities and GUI events. Then, static analysis is used to produce candidate races where false positives are ruled out by path-sensitive, backward symbolic execution. To detect non-conformant Android lifecycle handling, Zein et al. [24] presented a static code analysis tool that can verify that system resources - that are shared between different mobile apps such as Camera, GPS, Network connections...etc. - have been correctly initiated and released inside Android apps.

It can be observed from this part of the literature that none of the existing tools and approaches - given their varying accuracies in detecting Android lifecycle-related issues - presents an Android development-integrated support to assist developers discover problems at development time.

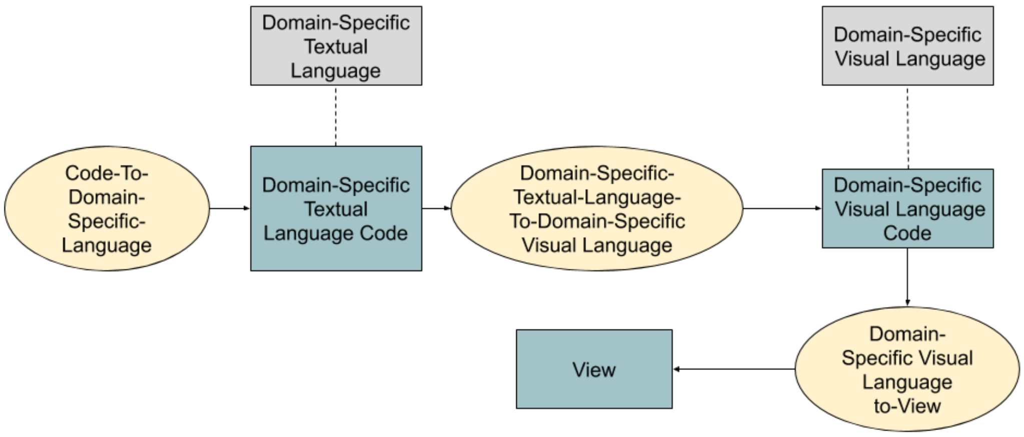

Domain-Specific Textual Language
Domain-Specific Visual Language
Code-To-Domain-Specific-Language
Domain-Specific Textual Language Code
Domain-Specific-Textual-Language-To-Domain-Specific Visual Language
Domain-Specific Visual Language Code
View
Domain-Specific Visual Language to-View

Fig. 1. Approach Architecture.

V. RESEARCH METHODOLOGY

To answer RQ1 and to address the problem of improper Android lifecycle handling, a model-driven dynamic multi-view and generic approach is presented to assist Android developers get a high-level view of their implementation of the Activity lifecycle. The approach implements a generic architecture that can be easily extended and utilizes two DSLs (a DSTL and a DSVL). The architecture is presented in Fig. 1.

Our choice of a multi-view approach was because it has been found that using multi-view based approaches for mobile development is beneficial while addressing multiple and specific concerns during app development [8] and with an extra level of abstraction, developers no longer need to deal with annotation and code and can concentrate on the model which reduces development complexity [9]. In general, model-driven development was found to have a high potential for accelerating software development for multiple platforms. It also increases standardizations in code and UI and considerably reduces time-to-market [10, 11, 12]. Also, it has been found that visual models (or graphical domain-specific languages) are generally more suited to cover a well-defined scope with sensible abstractions for domain concepts whereas textual domain-specific languages provide minor benefits to non-technical users because they feel like programming [12].

The approach defines a DSTL that can be used to describe an Activity lifecycle implementation. A static code analyzer is triggered when an Activity file is opened or selected. The code analyzer parses the Activity file source code and creates an internal model that conforms to the presented DSTL.

In addition, a DSVL is presented that can be used to describe instances of the DSTL for rendering purposes. After the intermediate DSTL model is generated, a DSTL-to-DSVL converter is used to transform the intermediate model into a DSVL model that conforms to the rules of the DSVL. Finally, the DSVL model is processed by a DSVL-to-View engine to render the visual model. The textual and visual DSLs allow the representation of information such as an Activity that is present in the app and the lifecycle callback methods implemented in it.

As a result, the developer will see two views of his Activity lifecycle implementation. The first view is the code that the developer has written. The second view is the visual view presented by our approach.

Several design patterns will be employed in the previous architecture to allow for adaptability and extensibility. For instance, the factory pattern will be used to allow for plugging in multiple/alternative implementations for the Code-To-Domain-Specific-Textual-Language converter and Domain-Specific-Visual-Language-To-View components.

The previous approach is a generic approach that can be adapted to any kind of mobile app's implementation. In order to answer RQ1, an Android specific version of both the textual and visual DSLs were defined and an implementation of the previous architecture, as an Android Studio plugin, was built to process implementations of the Android Activity lifecycle implementation. It should be noted that the previous architecture can have analogous versions and implementations for other types of mobile apps such as iOS, UWP and Xamarin.

A. Domain-Specific Languages

In our approach, two domain-specific languages were presented: a DSTL and a DSVL. The DSTL is defined using the UML modeling language and the DSVL is defined in terms of the visual notations that it supports. The DSTL definition can have implementations in any modeling language such as the Extensible Markup Language (XML) or JavaScript Object Notation (JSON) while the DSVL can have implementations in any general-purpose programming language such as Java, C#...etc.

1) Domain-Specific Textual Language: The DSTL presented here allows representing Android Activity classes, the implemented lifecycle handling callback methods, and any resource allocations or releases inside each of the lifecycle callback methods. The DSTL is defined as the following:

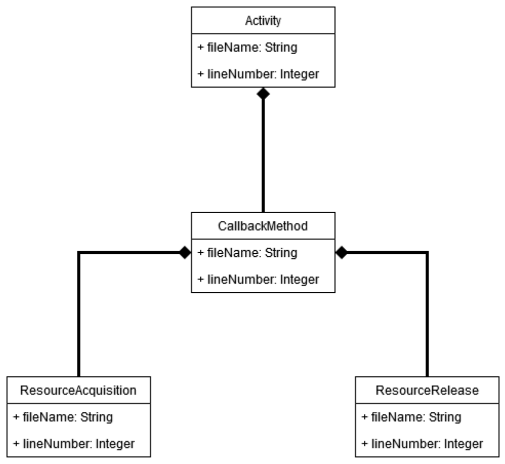


Fig. 2. Domain-Specific Textual Language.

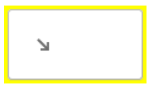
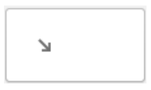
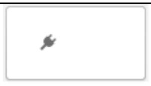
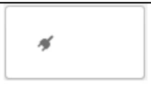
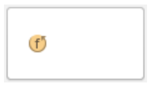
From Fig. 2, it can be observed that the language defines of the following entities:

1. **Activity**: this entity represents an Android Activity. From an implementation perspective, an instance of this element should exist only for the Activity classes that are registered in the app's AndroidManifest.xml file.
2. **CallbackMethod**: this entity represents lifecycle callback methods that are implemented inside the Activity class. Specifically, they are onCreate, onStart, onResume, onPause, onStop, onRestart, and onDestroy.
3. **ResourceAcquisition**: this entity represents an attempt to acquire a resource inside the lifecycle callback method or any other method in the subtree of the callback method.
4. **ResourceRelease**: this entity represents an attempt to release a resource inside the callback method or any other method in the subtree of the callback method.

It should be noted from the previous model that an Activity class can contain multiple CallbackMethod instances and a CallbackMethod instance can contain multiple ResourceAcquisition and/or ResourceRelease instances. In addition, each entity in the previous model contains a fileName and lineNumber attributes that point to the location in the app source code where the entity is defined or located.

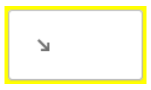
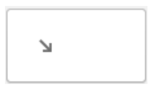
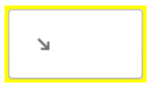
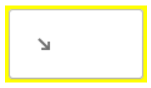
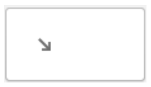
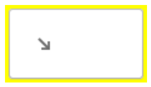
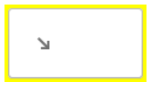
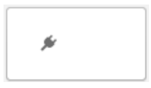
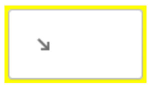
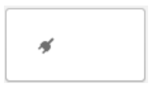
2) Domain-Specific Visual Language: The domain-specific visual language allows representing the Activity lifecycle states and the transitions between them. The following table illustrates the node types that the language defines:

TABLE I. Domain-Specific Visual Language Notations

Node	**Description**
	This node type represents a callback method that is implemented by the Activity class. The callback method represented by this node type should handle the transition of the Activity's state to the state represented by this node type.
	This node type represents a callback method that is not implemented by the Activity class.
	This node type represents a resource that has been allocated.
	This node type represents a resource that has been released.
	This node type represents a recursive callback method. This node type was introduced to eliminate the need to render the lifecycle graph as a circular graph and allows representing it as a tree.

The following table describes the edges between each two of the previous node types:

TABLE II. Domain-Specific Visual Language Notations

Source Node	**Target Node**	**Description**
		The line between the source node and target node means that the Activity may transition from the source state to the target state and only the source state transition is handled.
		The line between the source node and target node means that the Activity may transition from the source state to the target state and both state transitions are handled.
		The line between the source node and target node means that the Activity may transition from the source state to the target state and only the target state transition is handled.
		The line between any of the source nodes and the target node means that the callback method represented by the source node acquires a resource that is represented by the target node.
		The line between any of the source nodes and the target node means that the callback method represented by the source node releases a resource that is represented by the target node.

		The line between any of the source nodes and the target node means that the Activity may transition from the source state to a previous state that is represented by the target node.

VI. IMPLEMENTATION

The multi-view approach presented in this paper was implemented as an Android Studio plugin to allow developers to view their Android Activity lifecycle implementation. The following figures illustrate the plugin as seen by the Android developer:

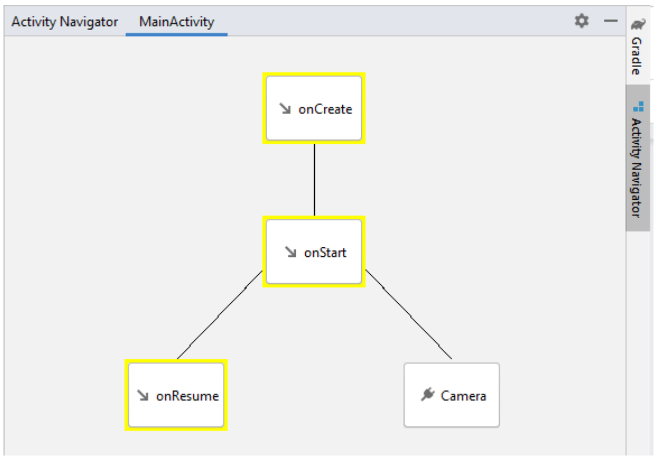


Fig. 3. Android Studio Plugin.

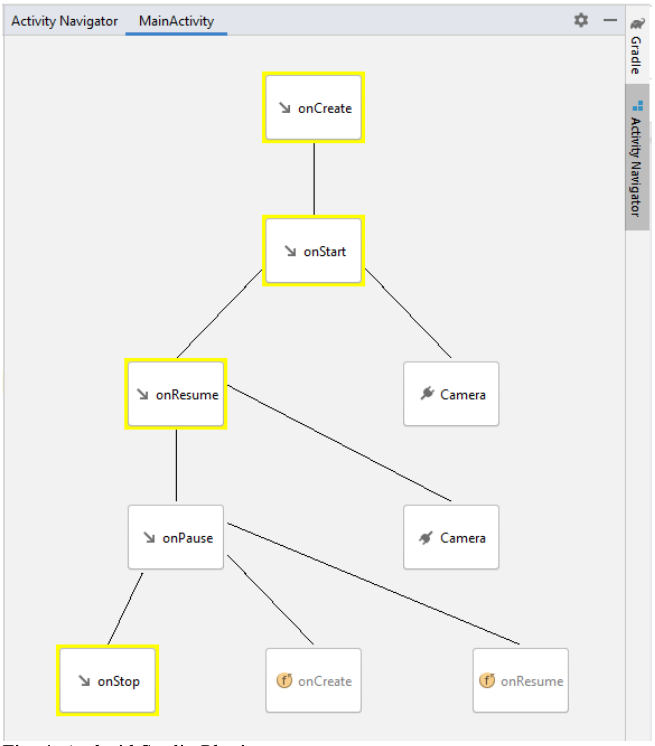


Fig. 4. Android Studio Plugin.

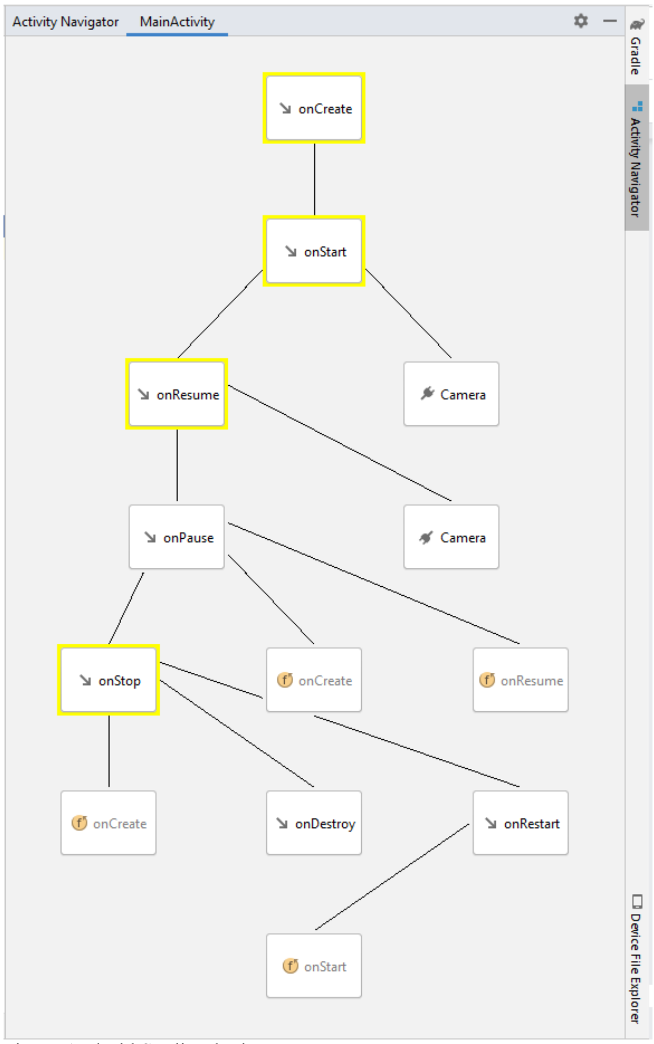


Fig. 5. Android Studio Plugin.

From the previous figures, it can be observed that the view is progressive, and the developer has the option to hide/show a subtree of the life cycle graph. The implementation includes additional features for the developers such as navigating to the implemented callback methods, adding unimplemented callback methods, and navigating to the allocated/released resources.

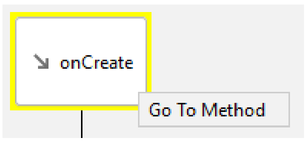


Fig. 6. Go to Implemented Callback Method.

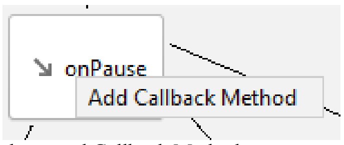


Fig. 7. Add Unimplemented Callback Method.

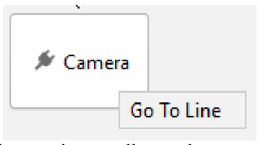


Fig. 8. Go to line which contains an allocated resource.

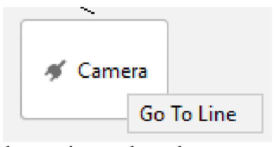


Fig. 9. Go to line which contains a released resource.

A. *Plugin Architecture*

Our approach was implemented as an Android Studio plugin[1]. The plugin was developed as a plugin to the IntelliJ Platform which allows it to be run on virtually any IDE that is developed on top of the IntelliJ Platform (e.g. IntelliJ IDEA, Android Studio...etc.). When developing for the IntelliJ platform, IntelliJ IDEA was used to develop the plugin. Development for the IntelliJ platform depends on two fundamental design patterns: Service-Locator and Message Bus. IntelliJ platform-compatible plugins can define services that are referenced by both the platform itself and other components inside the plugin. The Message Bus design pattern allows components to listen to specific types of messages and to send messages of specific types to all listeners that are loaded by any instance of the IntelliJ platform. Both design patterns allow for loose coupling between all components that are loaded by the instance of the platform. An instance of the IntelliJ platform is an IDE such as IntelliJ IDEA and Android Studio. The following figure illustrates the plugin architecture that conforms to the IntelliJ platform design principles.

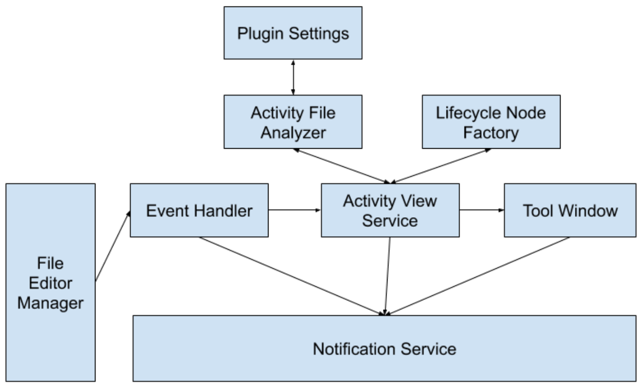


Fig. 10. IntelliJ Platform Plugin Architecture.

1) FileEditorManager: The FileEditorManager class in the IntelliJ platform is the class responsible for all UI-based file operations (e.g. open, select, close, edit, save) that the IDE user interacts with. The FileEditorManager class dispatches three types of file editing messages on the platform message bus: **fileOpened**, **fileClose**, and **selectionChanged**. The previous messages can be processed by any handler that implements the **FileEditorManagerListener** and has been registered on the platform message bus to listen for messages sent by the FileEditorManager.

2) EventHandler: The **FileEditorManagerListener** listener interface was implemented in an anonymous class and registered on the message bus to listen for messages that are sent by the FileEditorManager. The anonymous class, upon receiving a message sent on the bus by the FileEditorManager, passes the message to a class called **FileEditorManagerEventHandler** that is responsible for the following before passing the message to the inner components of the plugin:

1. Making sure that the file is in a valid state.
2. Making sure that the file opened by the user (i.e. developer) is an Activity file. This is done by locating the **AndroidManifest.xml** and searching for an Activity file registered with the same name as the opened, selected, or closed file.
3. Invokes the inner components of the plugin when the IDE is running in smart mode (i.e. when all the indexes that the IDE uses have been built). This choice has been made to abstract the need to access the real file system directly and instead use the IntelliJ platform API. Second, accessing built indexes has a significant performance gain when searching for files.

3) Activity View Service: The Activity View Service component is the main component responsible for maintaining the DSVL model of the Activity file and handling all events that are fired by the nodes in the visual model such as the event to expand a node, adding a callback method and navigating to a callback method. The Activity View Service depends on three components to perform its functions.

a) Activity File Analyzer: The Activity File Analyzer is the component responsible for analyzing the Activity file, parsed as a PSI element, and producing an instance model that conforms to the DSVL. The current implementation of the plugin uses an implementation of the analyzer that visits the subtrees inside the Activity class that start at the callback methods. The analyzer visits the entire subtree of the callback method looking for method calls that are identified as either a resource acquisition or resource release. For each method call in the subtree, static method call or instance method call, the analyzer produces a string that consists of the fully qualified class name and the method name and matches the result to the resource allocation and releases defined in the plugin settings. The analyzer produces an instance model that conforms to the DSTL.

b) Plugin Settings: The plugin settings contain two sections that define the method calls that are recognized as either resource allocation or resource release. Each entry in the settings file is defined as a key-value pair where the key is name of the resource and the value has the following format: <fully-qualified-class-name>.<method name>.

c) Lifecycle Node Factory: The LifecycleNodeFactory is the component responsible for transforming the DSTL model for the Activity class into an instance of the DSVL model that can be used to render the Activity lifecycle implementation.

d) Tool Window: The implemented tool window in the plugin is used to render the DSVL model instance that represents the Activity lifecycle implementation. The tool window is a tabbed window where each tab represents an opened Activity file in the FileEditorManager.

4) Notification Service: The plugin, in this current version, follows the fail-fast principle which is whenever an

[1] The source code can be found at https://github.com/tghanem/android-lifecycle-visualizer.

exception or an error occurs, the current processing thread fails immediately and returns back to the FileEditorManagerEventHandler anonymous class. Any failure in the current processing thread is reported as-is using a balloon notification dialog in the IDE. In addition, the failure is logged to the IDE event log.

VII. EVALUATION

A user evaluation of the Android Studio plugin was conducted for mobile app development. The primary aim of this user evaluation was to capture the user (i.e. developer) satisfaction of having a second view for the Android Activity lifecycle.

Towards this aim and in order to answer RQ2, the presented Android Studio plugin was evaluated by adopting the method of Barnett et al. [8]. A case study was conducted where the participants are Android mobile app developers with varying levels of experience. After the case study, the participants were asked to fill a questionnaire where they answered questions about their experience using the presented plugin. The questionnaire consists of 7 questions with 5-point Likert scale ranging from Strongly Disagree to Strongly Agree and 2 open-ended questions that capture the participants' experience of using the plugin. The questionnaire was designed by following the positive questionnaire design approach that is suggested by Sauro et al [25].

A. Case Study Setup: Each participant followed the following steps throughout the case study:

1. Fill a demographic survey which collects the participant's background information related to Android mobile app development.
2. Participate in a session to get introduced to the approach and the plugin that implements the approach.
3. Develop a simple Android application using the plugin.
4. Fill a post-case-study questionnaire to collect information and insight about the personal experience of using the plugin.

B. Participants: For user evaluation of the plugin, at least 6 Android app developers were needed. To recruit participants for the case study, 10 Android app developers were contacted through social media and direct face to face communication and only 6 of the participants completed the case study fully. The participants had a varying range of experience in Android app development.

C. Case Study App: The participants were asked to develop a simple Android application which consists of a single Android Activity where the app user can press and hold on a screen button to capture a video clip. Once the user releases the button, the app would capture the current device position and upload the bundle that consists of the video clip and the geographical location to an online web service. The participants were not required to complete the development of the app and produce a fully working application. The case study was focused on the development experience provided by the Android Studio plugin which provided a second view for the Android Activity.

D. Results: In order to answer RQ2 and understand the extent by which the presented approach affected the development of Android mobile apps by the developers, the evaluation aimed to collect the following data: understandability, accuracy, usability, and satisfaction of the presented approach through a post-case-study questionnaire that was filled by each participant. The answers to the demographic survey and post-case-study questionnaire were as the following:

1) Demographic Survey Results

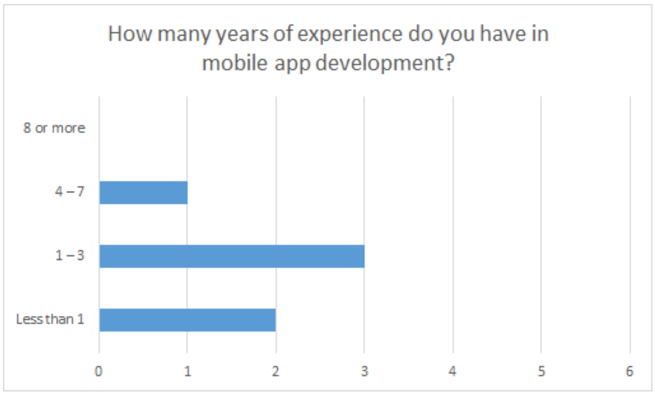


Fig. 11. How many years of experience do you have in mobile app development?

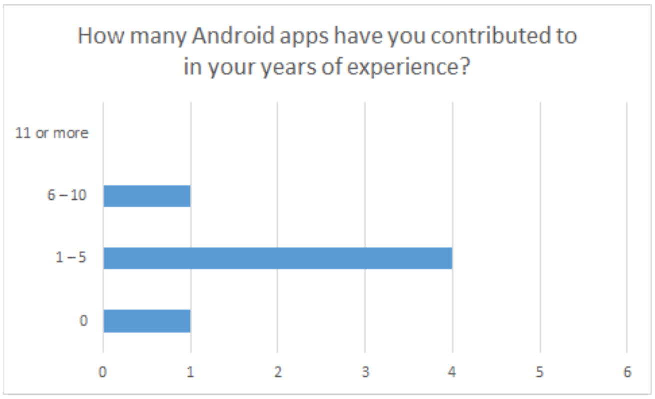


Fig. 12. How many Android apps have you contributed to in your years of experience?

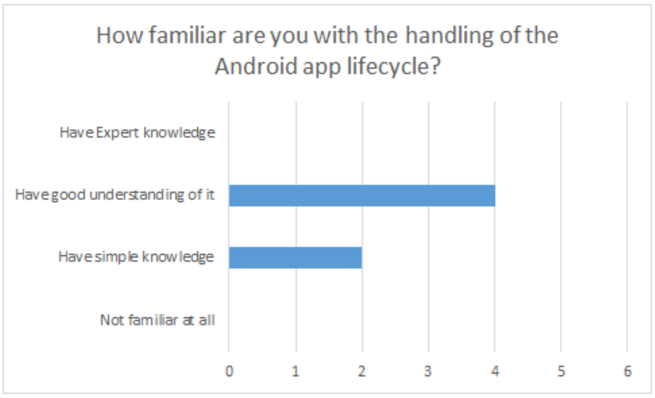


Fig. 13. How familiar are you with the handling of the Android app lifecycle?

2) Post-Case-Study Questionnaire

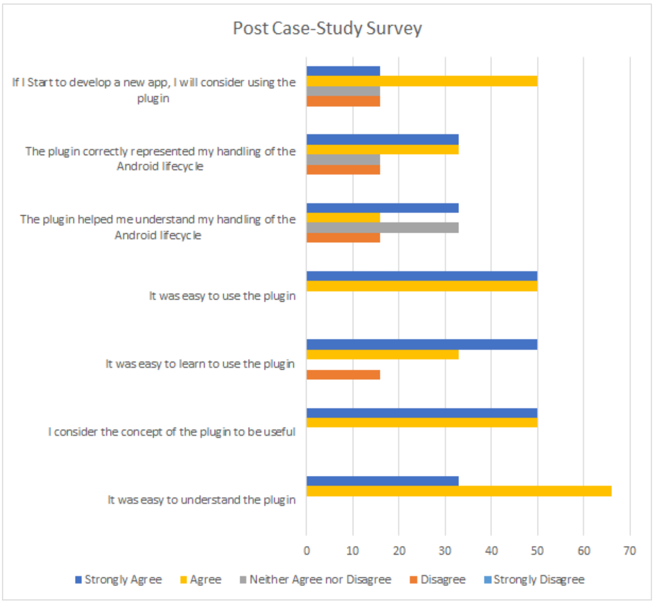


Fig. 14. Post-Case-Study Questionnaire Results.

When asked about whether they will consider using the plugin in future apps, 50% of the participants agreed that they will. Around 33% of the participants considered that the plugin correctly represented their handling of the Activity lifecycle. The participants didn’t report conclusive results when asked about whether the plugin helped them understand their handling of the Android lifecycle; 33% of the participants Strongly Agreed while 33% Neither Agree nor Disagree and one participant disagreed. The participants agreed that learning, understanding, and using the plugin were easy tasks. Only one participant disagreed that it was easy to use the plugin. Finally, the participants agreed that the concept of the plugin was a useful concept.

Further insights on the participants' experience and satisfaction of the presented approach can be observed from their answers to the open-ended questions.

a) What issues did you have when using the plugin?

1. It does not show correct behavior when resource allocations/releases function calls are nested inside other methods.
2. It crashed a couple of times, I had to restart the Activity or restart Android Studio.
3. Some methods used in various lifecycle methods were not shown in the plugin.
4. Plugin does not refresh immediately after implementing a new lifecycle method.
5. Recursive nodes are a bit confusing, in my opinion, I would like to see the recursion instead of showing a new node that is faded.
6. I had to re-open the Activity class every time I do a change to see the change reflect on the diagram.
7. Did not see a warning when using the camera from the button handler.

b) What changes do you suggest should be made to the plugin?

1. Warning of memory leaks, unclosed stream, and freeing of mobile resources.
2. The plugin should give more information about the Activity lifecycle; why we should handle every method and what are the cases that the app goes into mapped to those methods.
3. Double click to go to method implementation. This makes navigation between methods faster.
4. Add an option named “break here” where it runs the application in debug mode and hit a breaking point in the specified state.
5. During debugging, highlight the current state of the Activity. For example, if a breakpoint is hit inside a method that is called from any of the callback methods, highlight the corresponding node in the lifecycle view. This allows seeing the current state of the application without needing to review the stack trace.
6. Add an option to expand all the nodes in the graph in one click. Navigating through multiple unopened nodes is counterproductive and distracting.
7. I sometimes like to write To Dos inside a specific lifecycle method. It would be nice if the tool allows me to see the To Do comments and maybe click them for navigation.
8. Global state is needed in an Activity sometimes. It is helpful if I have the option to see this global state and how it is changed throughout the lifecycle of my Activity.

9. Add an option to implement all callback methods in the Activity class.
10. Keep track of resources within other classes, packages.
11. Support onRestoreInstanceState and onSaveInstanceState.
12. Support for Fragment lifecycle.
13. Use colors to show issues in the life cycle.
14. Warnings should be complete to show the correct location of the resource allocation/release.
15. Add support to correct the incorrect usage from the view.

The questionnaire also included further comments made by some of the participants:

1. The concept behind the plugin is useful for complex Activities, for simple activities, it seems unnecessary. I do not think an advanced developer will look for this diagram unless either the Activity is tool complex, or more features are introduced (to the plugin).
2. For advanced developers, the plugin is not very useful in helping them understand their handling of the Android lifecycle because they already understand the lifecycle. However, it will help beginner and intermediate developers.
3. The concept of the plugin is helpful. However, the current implementation is not. If more features are implemented and bugs are fixed and the view is dynamic, then I think the plugin can become really helpful.

E. Disucssion

From the collected information in the post-case-study questionnaire, the following points can be seen:

1. The participants' satisfaction ranged from Disagree to Strongly Agree.
2. The majority of the participants considered the concept of multi-view for the Activity lifecycle to be useful and that the plugin that implemented the concept was easy to understand, learn and use. Only one participant reported that learning the plugin was not an easy task.

The reason that some of the participants didn't consider or couldn't decide whether the plugin helped them understand their handle of the Activity lifecycle, correctly represented their handling of the Activity life cycle, or will consider using it in future applications can be inferred from their answers to the open-ended questions:

1. From the participants' answers to the first open-ended question which is about the issues they had while using the plugin, the participants' responses were either pointing to a bug, missing features, and unexpected behavior. In specific, the participants indicated that the plugin crashed couple of times where they had to restart Android Studio, resource allocation/release that occurred in nested methods (inside the callback methods) are not shown, and the recursive node in the view were confusing and should rather be replaced by a line back to the target node.
2. From the participants' answers to the second open-ended question where they had to suggest changes to be made to the plugin, it can be observed that developers expected the plugin to contain so many features: from warning of memory leaks, context information about each callback method, including TODO comments in each lifecycle method, tracking resources in other classes, packages...etc.

It can be inferred then from the answers to the two open-ended questions that Android developers expect that any plugin or tool that they use be fully featured and bug-free in order to be useful for them. The extra comments by the participants confirm the conclusions that were drawn from analyzing the responses to the open-ended question and comparing them with the previous questions.

From the previous discussion, the following conclusions can be made:

1. The concept of presenting a second view to Android developers about their Activity lifecycle handling, in addition to the code view, is a useful idea.
2. Any implementation that implements the concept of a second view to the Activity life cycle handle should contain many features and options and be bug-free in order to be endorsed and adopted by Android developers.

F. Threat to Validity

1) Internal Validity: The participants recruited for the case study have been recruited from software engineers at Zeva International. Their participation may have been biased in their responses because they knew the researchers which may have affected their responses. In order to remediate this threat, the researchers indicated to the participants that they should be as honest as possible in their responses to the questionnaire and that their responses whether positive or negative will be considered as empirical data and that the researchers have no interest in a specific type of response. The majority of the participants have less than 7 years of experience, contributed to less than 10 Android apps in their experience and have a good understanding of the Activity life cycle model. The case study didn't include participants developed more than 8 apps in their career, contributed to more than 11 or are experts in handling the Activity lifecycle. This has the effect of making the results of the case study biased towards novice and intermediate level Android developers.

2) External Validity: The number of participants that were involved in the case was 6 which is sufficient to draw the conclusions that we needed but not sufficient to draw statistically significant results. The participants were not required to develop a complete Android app and the software requirements were loosened as much as possible to keep the participants motivated to finish the case study. This has the effect of not reflecting the real nature of Android app development where developers are required to build bug-free apps and typically for a long period of time. If the plugin had been used in a real environment that involved a large number of Android developers for an extended period of time and where Android apps contain many Activity classes that are potentially complex, the results of the case study would have been more statistically reliable.

REFERENCES

[1] Tufail, H., Azam, F., Anwar, M. W., & Qasim, I. (2018, November). Model-Driven Development of Mobile Applications: A Systematic Literature Review. In 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON) (pp. 1165-1171). IEEE.

[2] https://developer.android.com/reference/android/app/Activity

[3] https://developer.android.com/guide/components/activities/activity-lifecycle

[4] Amalfitano, D., Riccio, V., Tramontana, P., & Fasolino, A. R. (2020). Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps. *IEEE Access*, *8*, 12217-12231.

[5] Toffalini, F., Sun, J., & Ochoa, M. (2019). Practical static analysis of context leaks in Android applications. *Software: Practice and Experience*, *49*(2), 233-251.

[6] Hoshieah, N., Zein, S., Salleh, N., & Grundy, J. (2019). A static analysis of android source code for lifecycle development usage patterns. *Journal of Computer Science*, *15*(1), 92-107.

[7] Riccio, V., Amalfitano, D., & Fasolino, A. R. (2018, July). Is this the lifecycle we really want? an automated black-box testing approach for Android activities. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (pp. 68-77).

[8] Barnett, S., Avazpour, I., Vasa, R., & Grundy, J. (2019). Supporting multi-view development for mobile applications. Journal of Computer Languages, 51, 88-96.

[9] Freitas, F., & Maia, P. H. M. (2016, November). JustModeling: An MDE Approach to Develop Android Business Applications. In 2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 48-55). IEEE.

[10] Vaupel, S., Taentzer, G., Gerlach, R., & Guckert, M. (2018). Model-driven development of mobile applications for Android and iOS supporting role-based app variability. *Software & Systems Modeling*, *17*(1), 35-63.

[11] Thu, E. E., & Nwe, N. (2017, June). Model driven development of mobile applications using drools knowledge-based rule. In *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)* (pp. 179-185). IEEE.

[12] Rieger, C., & Kuchen, H. (2018). A process-oriented modeling approach for graphical development of mobile business apps. Computer Languages, Systems & Structures, 53, 43-58.

[13] Li, X. S., Tao, X. P., Song, W., & Dong, K. (2018). Aocml: A domain-specific language for model-driven development of activity-oriented context-aware applications. Journal of Computer Science and Technology, 33(5), 900-917.

[14] Paspallis, N. (2019). An MDD□based method for building context□aware applications with high reusability. *Journal of Software: Evolution and Process*, *31*(11), e2200.

[15] Yigitbas, E., Jovanovikj, I., Biermeier, K. *et al.* Integrated model-driven development of self-adaptive user interfaces. *Softw Syst Model* (2020).

[16] Li, Y., Ouyang, J., Guo, S., & Mao, B. (2016, October). Data Flow Analysis on Android Platform with Fragment Lifecycle Modeling. In *International Conference on Security and Privacy in Communication Systems* (pp. 637-654). Springer, Cham.

[17] Junaid, M., Ming, J., & Kung, D. (2018, December). StateDroid: Stateful Detection of Stealthy Attacks in Android Apps via Horn-Clause Verification. In *Proceedings of the 34th Annual Computer Security Applications Conference* (pp. 198-209).

[18] Shao, Y., Wang, R., Chen, X., Azab, A. M., & Mao, Z. M. (2019, March). A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications. In Proceedings of the Fourteenth EuroSys Conference 2019 (pp. 1-14).

[19] Amalfitano, D., Riccio, V., Tramontana, P., & Fasolino, A. R. (2020). Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps. *IEEE Access*, *8*, 12217-12231.

[20] Toffalini, F., Sun, J., & Ochoa, M. (2019). Practical static analysis of context leaks in Android applications. *Software: Practice and Experience*, *49*(2), 233-251.

[21] Guo, C., Ye, Q., Dong, N., Bai, G., Dong, J. S., & Xu, J. (2016, November). Automatic construction of callback model for Android Application. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)* (pp. 231-234). IEEE

[22] Perez, D. D., & Le, W. (2019). Specifying callback control flow of mobile apps using Finite Automata. *IEEE Transactions on Software Engineering*.

[23] Hu, Y., & Neamtiu, I. (2018). Static detection of event-based races in android apps. *ACM SIGPLAN Notices*, *53*(2), 257-270.

[24] Zein, S., Salleh, N., & Grundy, J. (2017, May). Static analysis of android apps for lifecycle conformance. In 2017 8th International Conference on Information Technology (ICIT) (pp. 102-109). IEEE.

[25] J. Sauro, J.R. Lewis, When designing usability questionnaires, does it hurt to be positive? Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11, ACM, 2011, pp. 2215–2224.

[26] 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.

[27] Nirumand, A., Zamani, B., & Tork Ladani, B. (2019). VAnDroid: A framework for vulnerability analysis of Android applications using a model□driven reverse engineering technique. *Software: Practice and Experience*, *49*(1), 70-99.

[28] Junaid, M., Liu, D., & Kung, D. (2016). Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models. *computers & security*, *59*, 92-117.