Research Article

Porting Mobile Apps from iOS to Android: A Practical Experience

Khalid Lamhaddab1,2, Mohamed Lachgar3 and Khalid Elbaamrani1

1TIM Laboratory, ENSA, Cadi Ayyad University Marrakesh, Marrakesh, Morocco
2MDA Expert, MyAppConverter Ltd., London, UK
3LTI Laboratory, ENSA, University of Chouaib Doukkali El Jadida, El Jadida, Morocco

Correspondence should be addressed to Khalid Lamhaddab; k.lamhaddab@gmail.com

Received 19 April 2019; Revised 19 June 2019; Accepted 22 July 2019; Published 3 September 2019

1. Introduction

The recent rise of smartphones has triggered a revolution in mobile development. As a result of this incremental mobile innovation, new software engineering techniques, software documentation, and tools adapted to the mobile platform remain essential in order to help developers to better understand, analyze, and bootstrap porting mobile applications. In this paper, the authors propose a model-driven reverse-engineering approach based on static analysis, which describes a semantic metamodel of the iOS mobile application and extract design information (such as user interfaces, activity diagram, entities, framework and library dependencies) in order to generate the functional specification documentation and the Android UI skeleton. Thus, aiding the project team, who has in charge porting the app to another mobile platform, to agree upon a consensus on what has to be implemented and safe development cost by auto generating the Android UI skeleton project. To experiment this approach, the authors have implemented a tool called iSpecSnapshot. Moreover, they evaluate the performance of iSpecSnapshot by an experiment involving iOS applications that are ported to Android platform.

The growth in number of smartphone users worldwide is leading to an impressive growth in number of app downloads and also in terms of incomes generated by these apps. According to a recent statistical study [1], by 2021, over 353 billion of mobile apps will have been downloaded worldwide [1].

At present, there are over 2.2 million apps on the Apple store [2], and more than 3 million Android apps are on Google Play Store [3].

The key challenge that enterprise organizations face when starting their native mobile app development projects is what platform should they start with: iOS or Android?

In fact, both the two platforms have their fair share of pros and cons. The authors outline the key aspects, which can help to get a clear idea to decide which platform is suited to enterprise requirements. The choice comes down to 4 factors:

(i) Audience: Android is fast-growing and enjoying 75.27% of the share of whole mobile OS market compared to iOS, which ranks second at 22.74%. AppAnnie [4] reports that, for the first quarter of 2019, the Google Play Store pulled 22 billion apps worldwide. Comparatively, the App Store only drove 8 billion.

(ii) Monetization: While Android rakes in more downloads, iOS maintains a nearly 2x lead when it comes to gross consumer spend [4]. Accordingly, iOS customer segment comprises well-off users who are willing to pay extra for sustainable apps and services with advanced design, security and capabilities. The apple iOS monetization is built on a subscription model or in-app purchases. However, Android apps rely on an ad-based model.

(iii) Project timeline: Developing Android apps generally takes more time due to device fragmentation and the complexity involved in Android app development. On an average, Android app development is 30–40 percent slower than iOS. Although iOS apps are quicker to design, Apple has a strict

1.2

Hindawi
Mobile Information Systems
Volume 2019, Article ID 4324871, 29 pages
https://doi.org/10.1155/2019/4324871
Mobile Information Systems

approval process to control apps quality and their compliance to certain standards. In contrast, Google Play Store publication process implies less strict guidelines for publishing an app. Generally, an app review process takes a day or two to get approved and published within few hours.

(iv) Budget: The cost to develop a mobile app comes down to a range of factors. The scope and the complexity of the project are the most significant cost factors. There are other facets that incur the cost such as: Devices and OS versions support, Backend infrastructure and services integrations, Marketing effort, Team and cross department involvement, and Maintenance/upgrade.

Many startups who need to build a minimum viable product quickly and cheaply prefer to start with iOS platform. In the meantime, for startups that want to conquer the market shares and need to scale swiftly as well, porting an iOS app to Android is then a key strategy. With a wide Android user audience, they have certainly more monetization and investment opportunities. Alternatively, other enterprises prefer to build their mobile app using the progressive web app (PWA) strategy [5]. This path is more effective when it comes to code reusability, maintainability, cost efficiency, and audience reach (access via URL). Recently, progressive web apps can leverage some features similar to native apps such as send push notification, use touch gesture, access to phone’s accelerometer, and use of vibration. However, the major drawback of PWA is the behavior of performance and application. PWA will always lag behind native because the native APIs need to be developed before. In other words, PWA might not be the best option for developers if they look for a high performance or their apps use 3D graphics.

Porting iOS app to Android refers to converting or rewriting the app code according to the target platform, in order to enable it to run on different Android mobile devices and deal with the famous concern Android fragmentation [6]. Porting or reverse engineering an app has turned out to be a tedious and complex task, as the developers should have technical skills in both the source and target mobile platform. In fact, developers had to deal not only to adapt platform SDK and write specific code, but also to analyze and understand the functional logic behind a mobile app.

In a study carried out by Roehm et al. in [7], several developers act as end users and interact with the user interface to test the application’s behavior in order to find an entry point of the User Experience (UX) model. Others consider that source code is more credible than documentation; this was due to the lost or the outdatedness of the documentation.

In general, most mobile experts agree that the process of mobile app porting should take the following steps:

(i) Understanding the functional behavior
(ii) Analyzing UI/UX (user interface/user experience)
(iii) Technical assessment
(iv) Code conversion
(v) Testing and quality insurance of the produced source code
(vi) Iterative release

To meet the growing demands of mobile apps porting, dedicated software engineering techniques and reverse-engineering tools (see [8–10]) adapted to the mobile platform remain essential in order to help developers to better understand, analyze, and maintain mobile applications [11, 12].

Reverse engineering has many valuable benefits. The most principal advantages are as follows:

(i) Understanding of source code
(ii) Evolving and maintaining the software
(iii) Redocumenting software
(iv) Integration with legacy systems
(v) Migration of legacy systems to new recent technologies
(vi) Quality assessment

The main contribution of this paper consists of proposing a model-driven reverse-engineering (MDRE) approach, that considers both the UI part (storyboard workflow (Xcode Storyboard: https://developer.apple.com/xcode/interface-builder/)) and the source code of native iOS mobile application, to automatically extract specific models. These models contain UI/UX behavior and technical assessments that are needed to generate both the functional specification document of the iOS app and the corresponding Android UI skeleton. This is important to help designers to analyze systems and could be used to validate system requirements and accelerate development at a reasonable cost [13].

The proposed contributions are outlined below:

(i) A reverse-engineering technique that extracts automatically a semantic mobile app model, according to a predefined mobile metamodel. This metamodel includes more detailed information about the mobile app, such as screens, controllers, widgets, navigations, events, and actions.
(ii) An in-depth reporting analysis through the source code, to build a framework dependencies tree. In their case study, the authors are interested in iPhone SDK Framework (iOS SDK framework: https://developer.apple.com/documentation/).
(iii) iSpecSnapshot, which is a tool that analyzes the semantic model of the iOS mobile app and turns it into specific target models. The inferred models are transformed into functional spec document and Android UI skeleton, following a forward engineering technique [14]. The generated artifacts will help developers in their porting task to understand the source application and bootstrap the porting process.

This paper is organized as follows: Section 2 reports the most relevant mobile reverse-engineering approaches and compares them according to various criteria. Section 3
presents the empirical study design of our research. Section 4 describes the proposed MDRE approach for reverse-engineering iOS apps to Android. Section 5 describes iSpecSnap, which is based on a critical in-depth examination of works published over the last seven years regarding mobile reverse-engineering approaches. Finally, a discussion regarding the contributions achieved within this research and some possible challenges and perspectives for future work are presented in Section 7.

2. Related Work

The literature review presented is based on a critical in-depth investigation of works published over the last seven years with regard to mobile reverse-engineering approaches.

In [15], Frank et al. investigate how to reverse-engineer mobile application lifecycles by testing. The method is driven by developer's code and consists of four steps: (1) Full implementation of the lifecycle methods (pause, start, destroy). (2) Log Injection: printing some test logs inside the lifecycle methods. (3) Transition-trigger detection: identify a catalog of triggers and track the app behavior. (4) Application lifecycle rebuild: based on the logger data, the developer can rebuild a state model for the mobile app. The main weakness in their approach is that they oblige developer to hard code logs in order to achieve this task. Therefore, there is no automation effort such as implementing a plugin, which can perform source code analysis and detect the possible lifecycle methods and potential event behaviors.

In [16], Joorabchi and Mesbah propose a reverse-engineering technique based on dynamic analysis, in order to generate a state model of a running iOS application. This approach aims to capture the user interface states and transitions between them. The authors have endorsed their technique by a tool called ICRAWLIER. This approach, however, has not been able to describe correctly all the UI part characteristics (graphic information, UI classes, event types, etc.).

The study carried out by Yang et al. [17] addresses the issue of automated GUI-model generation for mobile applications. They introduce a grey-box approach for automatically extracting a model of a given mobile application. The approach aims to extract, using a static analysis, all the supported events by the GUI components of the application. Then, it uses a dynamic crawling process for reverse-engineering a model of the application, by applying—on the fly—the extracted events on the running application. The authors have developed a tool for Android platform called Orbit, which is a library of an action detector module based on WALA (T.J. Watson Libraries for Analysis WALA: https://github.com/wala/WALA), and a dynamic crawler built on top of Robotium (Robotium: https://github.com/RobotiumTech/robotium).

In [18], Salva and Zafimiharisoa propose a formal model inference approach for mobile application by automatic testing. The approach aims to store rich details about the encountered interfaces and help to reduce the application exploration, by using the Ant Colony Optimisation ACO strategy. The method is more suitable for performing an STS model (Symbolic Transition System) and test case generation.

In [19], Morgado et al. present an approach for testing mobile applications using reverse engineering and behavioral patterns. The aim of their work consists of defining a catalogue of mobile GUI behavioral patterns and using a hybrid reverse-engineering tool based on Yang et al.'s approach [17]. The tool helps to automatically explore mobile application, identifying patterns and applying the predefined test.

In [20], Nguyen and Csallner introduce a technique called REMAUI for inferring mobile application user interface code from screenshots or conceptual drawings. Hence, from a given input bitmap, REMAUI identifies user interface elements such as images, texts, containers, and lists via computer vision and optical character recognition (OCR) techniques. REMAUI merges OCR and computer vision results and in the merged data identifies the common structures, such as lists or images. REMAUI then recognizes aligned text blocks, groups them into a layout container, and exports the recognized text fragments as an Android resource file. A major weakness of this technique is that it is not well suited to recognize complex composite component such as menu/menu item, calendar, segmented controls, or even radio group. The core of the technique might have been more interesting, had the authors exploited the MDE paradigm during the OCR recognition phase. Thus, it will allow producing rich PSM (platform-specific model) models that describe in a more complete way the conceptual drawings.

Lamhaddad and Elbaamrani in [21] investigate how to use graph-based modeling in reverse engineering of mobile applications. The authors use a static analysis of the mobile app source code, in order to produce a representative graph model of the source app. The generated graph model can be transformed, via semantic rules, to another graph model according to a target mobile metamodel. Both the source and the target graph models are stored in a graph database. The study concludes that graph-based modeling provides advantages such as flexibility regarding the relationship between graph nodes, data consolidation, and the ease of navigation inside graph models thanks to a specific query language.

To overcome the challenge of how to avoid stateless event flow graph (EFG) of mobile application GUI, Amalfitano et al. in [22] present a GUI-driven testing framework for Android apps called MobiGUITAR. The framework is based on three steps: (1) reverse engineering the state-machine model from the running app and creating abstractions of the machine, (2) generating test cases, and (3) replaying the test cases. According to the study, MobiGUITAR produces several testing artifacts (such as Crash Report, Finite State-Machine FSM Model, GUI Sequences, and JUnit test cases).

In another study by Dugerdil and Sako in [23], the authors present a reverse-engineering process to recover the functional structure of mobile apps. The approach consists of three steps: (1) using code instrumentation by inserting extra statements in the source code, which helps to record events, (2) recording the execution trace in a flat file format, and (3) performing an offline analysis of the execution trace to retrieve the use cases of the mobile app using many views.
Another contribution by Salihu et al. [24] proposes a hybrid technique for reverse-engineering the GUI model from Android apps. The researchers have developed a tool called AMOGA that combines static and dynamic analysis. The tool performs a static analysis of mobile application binaries to automatically extract GUI information. Then, it instructs a dynamic crawling to explore and reverse-engineer a model of the mobile app.

Morgado and Paiva [25] developed a tool named iMPAct for supporting the automatic testing of mobile apps based on the presence of recurring behavior UI Pattern. The methodology used in this study was fully automated, and it is based on two steps: first, crawling the application by identifying a UI pattern and then interacting with it by applying a test strategy.

A field study approach is applied by Chen et al. [26] to investigate automating visual understanding of mobile UI design to generate a mobile GUI skeleton. Through their study, a neural machine translator is developed, which combines a vision convolutional neural network (CNN) and a recurrent neural network (RNN) [27] encoder/decoder. The translator consists of extracting visual features in UI design, encoding feature spatial layouts, and generating GUI skeletons. With the similar focus, another case study by Beltramelli [28] contributes to this line of research by presenting an approach based on convolutional and Recurrent neural networks combined with a specific DSL. The technique was trained on a relatively small dataset and requires more improvements to recognize vectorial representation. The author argues that generative adversarial networks (GAN)s [29] could potentially be used in combination with pix2code model to improve results.

A recent testing technique that is GUI exploration based is introduced by Amalfitano et al. [30]. The approach aims to exploit human involvement in the automated process to address the challenge of exploring Gate GUIs, which require to be exercised with particular input event sequences. The authors present jugular, a hybrid GUI exploration tool, combining automated GUI exploration with capture and replay by exploiting a machine-learning approach.

The previous studies were categorized according to the ontology presented initially by Cornelissen et al. [31] and refined according to the mobile reverse-engineering context by Morgado et al. [19]. The main aspects to classify the selected approaches included:

(i) Goal: the main aims of the studied approach (program comprehension, model recovery, pattern identification, verification and validation, etc.)

(ii) Target: the target platform (iOS, Android, Web) or the relevant input artifact (UI drawing, screenshots), which is concerned by the study.

(iii) Method: this refers to what type of analysis performed by each approach on the source artifacts, i.e., static, dynamic or hybrid.

(iv) Technique: this indicates the implemented technique for reverse engineering/comprehending the considered system (instrumentation, code injection, parsing, crawling, etc.).

(v) Extracted information: this specifies the key information extracted by the adopted method (runtime behavior, event sequence, UI states, crash detection, etc.).

(vi) Output: this indicates in which forms the results will be generated (call graph, finite state machine, report, test suite, etc.).

(vii) Validation: by what means the approach has been validated (case studies, comparison with other tools/approaches, evaluation).

Table 1 summarizes the result of the classification of the previous studies according to the ontology criteria presented by Morgado [32].

Based on this in-depth critical analysis of previous works, there remain many gaps that have not been tackled by recent research. In fact, the main challenges covered in these studies can be grouped in three topics: test automation, program comprehension, and mobile UI drawing. Considerable gap that warrant particular attention is that none of the previous research have addressed the full reverse engineering from one platform to another platform. In addition, most of these approaches, which rely on static analysis of the source code, have not take advantages of the use of the model-driven engineering (MDE) paradigms, in order to better comprehend the studied system. Only the attempt introduced by Lamhaddab and Elbaamrani [21] approached the reverse-engineering of mobile apps by focusing on graph-based models. We believe that the MDRE approach can bring a considerable gain regarding detection of behavior and UI patterns, GUI explorations, machine state identification, or event handling.

3. Empirical Study Design

In this section, the authors present the empirical study design of this research. They discuss the methods used for data collection and data analysis.

3.1. Data Collection. The first step of this research methodology is to perform a literature survey of the relevant published works regarding mobile reverse-engineering field, as presented in Section 2. The aim of this phase is to conduct a deep critical analysis of previous works and identify which current gaps remain in reverse-engineering mobile platform techniques to attempt to solve the gaps.

The second phase aims to collect the mobile developers’ needs. The objective of this study is to investigate the real challenges faced by mobile developers during the mobile porting process and to identify insight into the aspects and technical axis that must be covered when designing the proposed tool. Therefore, it was important to gather data from various stakeholders involved in mobile applications.
<table>
<thead>
<tr>
<th>Study</th>
<th>Goal</th>
<th>Target</th>
<th>Method</th>
<th>Technique</th>
<th>Extracted information</th>
<th>Output</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frank et al. [15]</td>
<td>Program Comprehension</td>
<td>iOS</td>
<td>Static</td>
<td>Instrumentation Code injection</td>
<td>Runtime behavior Event sequence Event handling</td>
<td>Call graph Report</td>
<td>Case study</td>
</tr>
<tr>
<td></td>
<td>Feature location</td>
<td>Android</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java ME</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joorabchi and Mesbah [16]</td>
<td>Model recovery</td>
<td>iOS</td>
<td>Dynamic</td>
<td>Crawling</td>
<td>Event handling</td>
<td>Finite state machine</td>
<td>Case study</td>
</tr>
<tr>
<td>Yang et al. [17]</td>
<td>Model recovery</td>
<td>Android</td>
<td>Hybrid</td>
<td>Crawling</td>
<td>Event handling</td>
<td>Finite state machine</td>
<td>Case study</td>
</tr>
<tr>
<td>Salva and Zafimiharisoa[18]</td>
<td>Verification and validation</td>
<td>Android</td>
<td>Dynamic</td>
<td>Crawling</td>
<td></td>
<td>Call graphFinite state machine Test suite</td>
<td>Comparison Evaluation</td>
</tr>
<tr>
<td></td>
<td>Model recovery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Morgado et al. [19], 2014</td>
<td>Verification and validation</td>
<td>Android</td>
<td>Hybrid</td>
<td>Crawling</td>
<td>Event handling</td>
<td>Call graph Report Test suite</td>
<td>Case study</td>
</tr>
<tr>
<td></td>
<td>Pattern identification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nguyen and Csallner [20]</td>
<td>User interface identification</td>
<td>Mobile</td>
<td>Static</td>
<td>Optical character recognition OCR Event handling</td>
<td></td>
<td>Android UI skeleton</td>
<td>Case study</td>
</tr>
<tr>
<td></td>
<td>Mobile UI design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lamhaddab and Elbaamrani[21]</td>
<td>Porting</td>
<td>iOS</td>
<td>Static</td>
<td>Parsing</td>
<td>Graph modeling Model transformation</td>
<td>Graph models for source platform (iOS) Graph models for target platform (Android)</td>
<td>Comparison Evaluation</td>
</tr>
<tr>
<td>AmalFitano et al. [22]</td>
<td>Verification and validation</td>
<td>Android</td>
<td>Dynamic</td>
<td>Ripping</td>
<td>Crash detection Bugs</td>
<td>Call graph Report Test suite</td>
<td>Case study</td>
</tr>
<tr>
<td></td>
<td>Model recovery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dugerdil and Sako [23]</td>
<td>Program comprehension Maintenance</td>
<td>iOS</td>
<td>Static</td>
<td>Instrumentation Code injection</td>
<td>Runtime behavior Event sequence Event handling</td>
<td>Call graph Report Test suite</td>
<td>Case study</td>
</tr>
<tr>
<td>Salihu et al. [24]</td>
<td>Model recovery</td>
<td>Android</td>
<td>Hybrid</td>
<td>Crawling</td>
<td>Event handling</td>
<td>Finite state machine</td>
<td>Comparison Evaluation</td>
</tr>
</tbody>
</table>
development. Interview participants were recruited through the subscriber database of MyAppConverter (MyApp-Converter: http://www.myappconverter.com) platform. Therefore, candidates have been selected via an e-mail campaign that highlights the road map of the new tool and which encourage subscribers to participate in the survey. The authors have selected 20 Android developers and 15 iOS developers with 2 to 5 years of experience. In addition, they have targeted 10 product owners who are not necessarily developers but rather business analysts. To approve the interview questionnaire, the authors have hired a small team from MyAppConverter composed of two senior mobile developers (iOS & Android) and one functional analyst lead. They recruited a total of 45 interview participants and scheduled 15 to 20 minutes conference calls for conducting interviews based on their availability. It should be noted that all the participants were ensured about the confidentiality of their personal information and data.

Moreover, three variants of questionnaires were administered to three categories: iOS developers, Android developers, and product owners. The purposes of these questionnaires are to identify the best practices and techniques widely used to design the UI part of iOS applications, list the components that present difficulties when they are ported to Android, and define the key information that will help in the functional understanding during the porting process.

3.2. Data Analysis. After collecting the practitioners’ responses, the authors analyzed carefully the data in order to determine the proposed tool's features. It is worth noting that real-world practitioners support the research as the majority of the 45 interviewees, agreed that implementing a tool which can reverse-engineer the functional specifications and port the UI storyboard, and could contribute substantially on time and cost of mobile porting project.

With regard to the questionnaire that targets product owners, 70% of the respondents agreed that the proposed tool should represent the functional requirements as use cases format. In use cases template, the functional requirements are understood in the context of user action, which typically avoid a lot of ambiguity that makes its way into an out-of-context list of system. However, 30% of the respondents prefer to represent the functional spec in an agile form such as user stories. They argued that the user story form is highly effective at capturing and linking user goals, functional requirements, and business benefits. Overall, 95% of the interviewees agreed that the functional spec document should include information about: project team, build information, dependencies, constraints, business class model, entity model, screens details, activity diagram, and business rules.

The respondents of the questionnaire that was targeted the iOS developers reflected that a high percentage (85% of them) use storyboard and xib files to design the UI layer. They argue that the Interface Builder IDE within Xcode makes it much easier to design UI and flows without writing any code. In addition, storyboards are highly recommended with multiple interconnected view controllers and eliminate boilerplate code to perform transition between view controllers. However, 15% of the respondents build UI programmatically by writing Objective-C or Swift code. They argue that mastering the coding of iOS UI allows addressing views with complex or dynamic layouts, customizing transition effects between view controllers, and refactoring specific views in a reusable fashion. 65% used autolayout feature in order to define constraints to control how user interface will adapt on different screen sizes, when device rotated, or when the app run in different locales. Overall, 47% prefer to use third libraries to customize UI components instead of iOS

<table>
<thead>
<tr>
<th>Study</th>
<th>Goal</th>
<th>Target</th>
<th>Method</th>
<th>Technique</th>
<th>Extracted information</th>
<th>Output</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen et al. [26]</td>
<td>User interface</td>
<td>Mobile UI design</td>
<td>Static</td>
<td>Neural machine translator</td>
<td>User interface widgets</td>
<td>Android UI skeleton</td>
<td>Case study Evaluation</td>
</tr>
<tr>
<td></td>
<td>identification</td>
<td></td>
<td></td>
<td>Convolutional neural network (CNN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Recurrent neural network (RNN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Convolutional neural network (CNN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Recurrent neural network (RNN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DSL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beltramelli [28]</td>
<td>User interface</td>
<td>Mobile UI design</td>
<td>Static</td>
<td>Convolutional neural network (CNN)</td>
<td>User Interface Widgets</td>
<td>Android UI skeleton</td>
<td>Case study</td>
</tr>
<tr>
<td></td>
<td>identification</td>
<td></td>
<td></td>
<td>Recurrent neural network (RNN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Call graph</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amalfitano et al. [30]</td>
<td>Verification and</td>
<td>Android</td>
<td>Hybrid</td>
<td>Crawling Instrumentation</td>
<td>Call graph</td>
<td>Finite state machine</td>
<td>Case study</td>
</tr>
<tr>
<td></td>
<td>validation and</td>
<td></td>
<td></td>
<td>Input event sequence</td>
<td></td>
<td>Test suite</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Model Recovery</td>
<td></td>
<td></td>
<td>Machine learning</td>
<td></td>
<td>Report</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Continued.
UI Kit framework. Additionally, 90% used xib to design static table view cell and displaying dynamic data. Furthermore, they used code to define table view data source (UITableViewDataSource: https://developer.apple.com/documentation/uikit/uitableviewdatasource) and UITableViewDelegate (UITableViewDelegate: https://developer.apple.com/documentation/uikit/uitableviewdelegate) in order to manage cell selection and other features related to displaying the data.

With regard to the questionnaire that targets the Android developers, 90% of the respondents used xml layout to design UI prototyping. Overall, 60% used small width qualifier in order to adapt layout to respond gracefully to different screen sizes and orientations. However, 40% agreed that it is preferable to use the constraint layout to create a responsive UI. In addition, a high percentage (92%) of the respondents recommend modularizing UI components with fragments (Android Fragments: https://developer.android.com). Overall, 100% agreed that it will be a tedious task to port some iOS UI components such as Attributed TextView, TableView, Collection View Controller, Page View Controller, Split View Controller, and Collection Reusable View. Moreover, all the interviewees admit that there is an additional effort to adapt iOS graphic assets for Android. The best solution is to scale assets to correct size for relevant device screen. Additionally, 90% of the respondents confirm that it is not obvious to find an equivalent for custom UI components; in most cases, developers are limited to using the closest Android UI standard components based on Google’s guidelines.

As running examples, the authors have chosen three open sourced projects from Apple sample code Library and Github. Table 2 shows resource link details for each running example (ID, name, and resource link).

Table 3 reports some metrics of each running example: number of the source code files (header and main files), size of the source files, number of used storyboard/xib files, and the number of the static UI elements.

4. Proposed Approach

This section outlines the model-driven reverse-engineering (MDRE) approach followed in iSpecSnapshot. The authors’ vision is to approach MDRE from a new context, which is mobile development. They will try to provide answers to enrich the reverse-engineering literature, with a hot topic for mobile development, with a high percentage (92%) of the respondents recommend modularizing UI components with fragments (Android Fragments: https://developer.android.com). Overall, 100% agreed that it will be a tedious task to port some iOS UI components such as Attributed TextView, TableView, Collection View Controller, Page View Controller, Split View Controller, and Collection Reusable View. Moreover, all the interviewees admit that there is an additional effort to adapt iOS graphic assets for Android. The best solution is to scale assets to correct size for relevant device screen. Additionally, all the respondents confirm that it is not obvious to find an equivalent for custom UI components; in most cases, developers are limited to using the closest Android UI standard components based on Google’s guidelines.

Specifically, in their method, the authors proceed in the same way as the Fleurey et al. [33] by adopting their MDRE steps with some alterations: code to PSM, PSM to PIM, PIM to PSM, and PSM to code. The particularity of the authors’ approach is that all the adopted metamodels are based on KDM and ASTM standards (see Appendix B). Hence, the proposed process is depicted in Figure 1 and consists of four main steps:

(i) Model discovery (code to PSM)
(ii) Model semantic extraction (PSM to PIM)
(iii) Model transformation (PIM to PSM)
(iv) Model to code generation (PSM to code)

4.1. Step 1: Model Discovery. Model discovery is a parsing step, which aims to parse the source mobile app artifacts in order to produce a set of representative models. These so-called initial models have a strong dependency with the components of the source app; they are commonly described by the OMG as platform-specific models (PSMs) (see Appendix A). The implementation of this step depends on the technology of the source mobile platform. It is based mainly on the static analysis to automatically generate PSM models from source code artifacts. Typically, an iOS app is composed by: programming language files (.h, .m, Swift, .cpp), platform-specific files (storyboard, .pch, .plist, .string), asset files (.png, .jpeg, .bmp), and data files (.xcdatamodeld, .xml, .json).

In the initial stage, it is wise to first define all the metamodels that describe the source mobile platform. Indeed, as mentioned in Figure 2, the authors distinguish four variants of metamodels: technological, infrastructure, applicative, and data metamodels.

The technological metamodel describes the programming languages used in the source mobile platform. It inherits mainly from the OMG ASTM metamodel. For instance, for the iOS mobile platform, the authors define ASTM metamodels for the commonly used programming languages such as: C, C++, Objective-C, and Swift. Figure 3 illustrates an excerpt of the Objective-C metamodel (metamodeling of the Objective-C block expression (Objective-C block: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html#apple_ref/doc/uid/TP40007502-CH1-SW1)).

The infrastructure metamodel defines the tree structure and relative paths of the source mobile app (folders, sub folders, files, assets, resources, etc.). It inherits mainly from the OMG KDM metamodel. Figure 4 shows an overview of the iOS xcode project metamodel.

The applicative metamodel defines mainly specific information regarding the source mobile platform including build settings, dependencies, compiler, executable name, and static UI workflow. It is designed according to the official iOS documentation (UI Kit official documentation: https://developer.apple.com/documentation/uid). Figure 5 shows an overview of the metamodel specific to the UI part (xib file).

The model discovery step is performed through platform parsers, which are specific to the source platform. The authors have set up four kinds of parsers:

(i) Infrastructure parsers allow generating a physical model of the source application. This model contains all the tree structure and relative paths of the
source mobile app (folders, sub folders, files, assets, resources, etc.). The generated model is conformed to the Xcode Project KDM metamodel.

(ii) Technological parsers allow generating abstract syntax models of all the programming language files used in the source application (e.g., C, C++, Objective-C, and Swift). At this stage, all the generated models conform to the technological ASTM metamodels.

(iii) Applicative parsers allow building applicative models, which are specific to some key artifact of the source platform (e.g., workspace file, xcodeproj file, pch file, plist file, xib file, Pod file, etc.). Generally these artifacts include information about the source platform, namely, build settings, dependencies, compiler, executable name, and static UI workflow. These models (e.g., Algorithm 1) are designed according to the applicative metamodels.

(iv) Data parsers allow building data models, which are statically stored in the file system (e.g., xdatamodel file, json file, xml file, etc.). These models conform to typical metamodels.

4.2. Step 2: Semantic Extraction. The semantic extraction step is considered as a halfway point between code-to-model and model-to-model steps. In fact, it aims to analyze the PSM models in order to extract some useful semantic information, which is required to build the PIM model. The extraction process is designed to be iterative and supports both technological and applicative models. This process is driven by a set of semantic extractors that identify, collect, and consolidate all the semantic data required to build the semantic mobile model PIM which is an instance of the semantic mobile metamodel. The authors have opted for a generic metamodel, which represents, in a semantic manner, any mobile application. The proposed metamodel is the result of a close collaboration with two senior mobile

---

**Table 2: Running examples: open-source native iOS apps.**

<table>
<thead>
<tr>
<th>ID</th>
<th>Resource name</th>
<th>Resource source link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TheSimpliestTodoList</td>
<td><a href="https://github.com/altaibayar/TheSimpliestTodoList">https://github.com/altaibayar/TheSimpliestTodoList</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="https://developer.apple.com/library/content/samplecode/uicatalog/Intro.html#//apple_ref/doc/uid/DTS40007710">https://developer.apple.com/library/content/samplecode/uicatalog/Intro.html#//apple_ref/doc/uid/DTS40007710</a></td>
</tr>
<tr>
<td>2</td>
<td>UIKitCatalogue</td>
<td><a href="https://developer.apple.com/library/content/samplecode/Tabster/Intro.html#//apple_ref/doc/uid/DTS40011213">https://developer.apple.com/library/content/samplecode/Tabster/Intro.html#//apple_ref/doc/uid/DTS40011213</a></td>
</tr>
<tr>
<td>3</td>
<td>Tabster</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3: Metrics of the running examples.**

<table>
<thead>
<tr>
<th>ID</th>
<th>No. of source files (h/m)</th>
<th>Size of source file (KB)</th>
<th>No. of storyboard (xib file)</th>
<th>No. of UI elements</th>
<th>Xcode build (SDK)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>30</td>
<td>1</td>
<td>26</td>
<td>7.2/9.0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>192</td>
<td>2</td>
<td>157</td>
<td>7.2/9.0</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>22</td>
<td>3</td>
<td>31</td>
<td>7.2/9.0</td>
</tr>
</tbody>
</table>

---

Figure 1: Main steps of the proposed MDRE approach to reverse-engineer a mobile app: code to PSM, PSM to PIM, PIM to PSM, and PSM to code.
Figure 2: First step of the proposed MDRE approach: model discovery (code to PSM).

Figure 3: Technological metamodel: an excerpt of the AST metamodel corresponding to the Objective-C block expression. The key entities are ObjcBlockExpression, Expression, Statement, FormalParameterDefinition, TypeReference, AccessKind, AstmObject, KDMEntity, etc.
developers (iOS and Android) from the MyAppConverter team. The metamodel is thus refined and readjusted during the private beta test of iSpecSnapshot. The metamodel inherits mainly from the KDM standard (Figure 6). The semantic mobile metamodel is briefly outlined below:

(i) A mobile application entity (MobileApp) has some information such as app name, executable name, version, author, package, etc. It is composed of several screens (AbstractScreen inherits from Screen which is described in the KDM metamodel). MobileApp contains information about orientation parameters also (ApplicationOrientation).

(ii) A screen has a single controller, which is used to manage the screen’s behavior (AbstractController...
(i) A controller, which refers to View Controller for iOS and Activity for Android, groups methods that implement the behavior of the actions. It also defines methods for managing the controller’s lifecycles.

(ii) The UI component, which is identified as an AbstractView entity, inherits from UIElement, which is described in the KDM metamodel. It is described by the following attributes: id, width, height, marginTop, marginBottom, etc. The UI component is likely to handle events.

(iii) The event entity can trigger an action (AbstractAction). The result of the action may lead to a redirection to the same screen or to another screen or even to another application.

This stage includes also the following substeps: PSM model exploring, PSM model querying, and PIM model generating. To build the PIM model, the extractors mentioned above, seek to identify the following information:

(i) General information: app name, icon, app package, version, and author

(ii) Platform information: build, device orientation, linked frameworks, libraries, and localization

(iii) Use cases information: launch screen, screen workflow, controllers, app lifecycle behavior, navigations, events, actions, and service flow

(iv) UI components properties: size, position, offset, range, color, state, transition, and effects

The resulting PIM model is in accordance with the previous semantic mobile metamodel.

4.3. Step 3: Model Transformation. The goal of this model-to-model M2M transformation phase is to transform the PIM model generated at the end of the previous step into a PSM specific to the target platform. In the literature, model transformation can be performed using various languages: declarative (e.g., QVT [34]), imperative (e.g., Java) or hybrid (e.g., ATL [35]). In the case of the current approach, the authors have opted for Java as an imperative language to drive the transformation stage. Their choice makes sense for many technical reasons, namely, in terms of design,
refactoring, performance, and integration. The process is obviously driven by metamodels. It is carried out explicitly by a set of transformers likely imperative language (e.g., Java), which allow exploring the PIM model, querying, computing, and generating the target PSM. In other words, the transformation rule was designed as a FROM-TO dipole. The rule is triggered when the FROM node fulfills several conditions related to the TO node. The rule logic may include special treatments and data computation depending on the target platform. The process is recursive until the PSM model is built.

To sum up, the model-to-model process involves the following workflow: model exploration, model querying, model computation, and PSM generation. With regard to the studied scope, the authors have described the target metamodels which are specific to the Android platform, while specifying the following areas: applicative and documentation. The Applicative metamodel defines mainly specific information regarding the target mobile platform including (build settings, dependencies, compiler, executable name, and static UI workflow). It is designed in accordance with the official Android documentation (Android UI & navigation official documentation: https://developer.android.com/guide/topics/ui/). An overview of the Android applicative metamodel is illustrated in Figure 7.

Algorithm 2 visualizes an example of the transformed PIM model to Android UI model (PIM to PSM):

The documentation metamodel describes all the needed information to build the structure of a functional specification document. The authors have opted for the use case format as suggested by the interviewees in the empirical study section. The document includes the following information: app name, app icon, app content, build information, activity diagram, business class model, entity class model, screens, screen components, and rule actions. An overview of the documentation metamodel is illustrated in Figure 8.

Additionally, the transformation process includes a complementary key substep, which is the mapping between the two platforms. The authors distinguish two types of mapping:

(i) Semantic mapping: It aims to define a semantic correspondence between the applicative metamodels for both source and target mobile platforms. For instance, it gives answers to how can seek to address matching between mobile platform UI components. Table 4 shows a list of the semantic mapping between iOS UI components and Android UI Components.
At this stage, the authors also define several transformation rules in order to match from a semantic point of view the UI inconsistencies between the two platforms, namely, text alignment, navigation bar, font family, button styles, controls, icons ... (Table 5).

Moreover, the authors set up a specific transformation rule to support Android multidevices. iOS devices come in a variety of screen sizes and can be used in either portrait or landscape orientation (Table 6). It was noted that, in iOS, the position of all UI components are absolute on a device metric coordinate system base, except for autolayout feature. Using autolayout, allows defining rules (known as constraints) that govern the content in the iOS app.

Android runs also on a variety of devices that have different screen sizes and pixel densities (Pixel density on Android: https://material.io/design/layout/pixel-density.html#pixel-density-on-android) (Table 7).

The dedicated transformation rule will be based on an absolute metric system. To assess this, the authors used the smallest width qualifier (the smallest width qualifier specifies the smallest of the screen’s two sides, regardless of the device’s current orientation: https://developer.android.com/training/multiscreen/screensizes#TaskUseSWQuali). The “smallest width” screen size qualifier allows providing alternative layouts for screens that have a minimum width measured in density pixels (see Table 8).

The authors have implemented a cross-multiplication rule, in order to identify the width rate and the height rate for the two device platforms. They have considered devices that have metrics closer to each other (e.g., iPhone 6 vs. Samsung S5). The rule is as follows:

\[
\text{width rate} = \frac{\text{Android device width}}{\text{iOS device width}}
\]

\[
\text{height rate} = \frac{\text{Android device height}}{\text{iOS device height}}
\]

(ii) Technological mapping: it aims to establish a mapping knowledge base [21]. The base is designed as a node graph, which manages the relationship between the SDKs API of the source and target mobile platforms (see Figure 9). The analysis of the source ASTM model allows generating a dependency report of the used APIs throughout the source mobile application. This report helps developers to drive the choice of which target API to use during the porting process.

4.4. Step 4: Code Generation. This last step of the reverse-engineering process is considered as a model to text M2T transformation. It aims to generate code or textual artifacts from PSM models. The process is broken down into several generators. Each generator is considered as a template engine that serves a specific domain of the targeted platform.

The authors have opted for the PSM to code strategy, which is widely considered to be the most suitable for
describing and representing the app requirements in a given development platform. However, authors in [36] highlighted other code-generation strategies when it comes to MDD approach, such as PIM-to-PSM-to-Native Code, PIM-to-Native Code, PSM-to-Native Code, PIM-to-Cross Platform Code, PIM-to-Cross Platform Framework, and SM-to-Cross Platform Code. The generators are basically based on Xpand (Eclipse Xpand: http://wiki.eclipse.org/Xpand), which is a Model to Text Language (MTL) [37] dedicated for code generation based on EMF models. Xpand provides interesting features such as polymorphic template invocation, aspect oriented programming, functional extensions, and model validation. There are also other M2T alternative tools like Acceleo (Eclipse Acceleo: http://www.eclipse.org/acceleo/) and JET (Eclipse JET: http://www.eclipse.org/modeling/m2t/downloads/?project=jet#archives), which provide code-generation framework and facilities that are used by EMF. The basic idea of generators is to refine and transform PSM models into code. The templates contain fragments of the target text and pieces of code that are replaced with information derived from the PSM model. The authors have defined several templates for each application domain:

Algorithm 2: PIM to PSM transformation: an excerpt of the inferred Android UI PSM model.
(i) Templates for Mobile App physical structure (Android studio structure)
(ii) Templates for target language (Java class)
(iii) Templates for UI part (xml layouts, drawables)
(iv) Templates for specific assets (manifest, gradle config, values, strings, and build settings)
(v) Templates for documentation (functional spec docs and readme file)
(vi) Templates for data (json, csv, xml, and SQLite)

Figure 10 illustrates an overview of xpand templates designed for functional spec document. Figure 11 illustrates an overview of xpand templates designed for Android UI components. Figure 12 outlines an overview of the result of PSM to code step which is performed on the functional spec PSM model. The generated source codes for the three running examples are shared publicly on a dedicated Gitlab repository (Tabster (Tabster Android UI project: https://gitlab.com/ispecsnapshot/tabster), TheSimpliestTodoList (TheSimpliestTodoList Android UI project: https://gitlab.com/ispecsnapshot/the simplesstodolist), and UICatalog (UICatalog Android UI project: https://gitlab.com/ispecsnapshot/uikitcatalog)).

4.5. Discussion. In the literature, Brunlière et al. in [38] have drawn the authors’ attention to a “Must Have” characteristic that each MDRE approach should abide by. The authors believe that these characteristics seem to be well founded, especially if a sustainable and scalable approach is to be ensured. These criteria are as follows:

(i) Genericity: an MDRE approach should rely on OMG metamodels standards and customizable model-based components
(ii) Extensibility: an MDRE approach should guarantee a separation between the models (as input or output elements) and the actions and treatment of each stage of the reverse-engineering process
(iii) Application domain: the Scope of the approach concerns specific or generic purpose
(iv) Automation: a reverse-engineering process should be totally or partially automated

(v) Full/Partial coverage: the modeling should cover all the representative artifacts of the system and also maintain a semantic link between them

4.5.1. Genericity.

The proposed approach provides a core metamodel based on the OMG standards KDM and ASTM specification. Indeed, for each MDRE step, the authors have described a set of metamodels depending on the reverse-engineering purpose, for example:

(i) Used metamodels for model discovery step: Xcode Project metamodel, Objective-C metamodel, Swift metamodel, Storyboard metamodel, Xib metamodel, Plist metamodel, andxcdatamodel metamodel

(ii) Used metamodels for semantic extraction step: AndroidApp metamodel, AndroidProjectStructure metamodel, Java metamodel, and Functional spec document metamodel

Based on the list of criteria described above, the authors try to give answers on how their approach remains efficient regarding the MDRE field.

4.5.1. Genericity. The proposed approach provides a core metamodel based on the OMG standards KDM and ASTM specification. Indeed, for each MDRE step, the authors have described a set of metamodels depending on the reverse-engineering purpose, for example:

(i) Used metamodels for model discovery step: Xcode Project metamodel, Objective-C metamodel, Swift metamodel, Storyboard metamodel, Xib metamodel, Plist metamodel, and xcdatamodel metamodel

(ii) Used metamodels for semantic extraction step: semantic metamodel

(iii) Used metamodels for model transformation step: AndroidApp metamodel, AndroidProjectStructure metamodel, Java metamodel, and Functional spec document metamodel

Based on the list of criteria described above, the authors try to give answers on how their approach remains efficient regarding the MDRE field.

**Table 4: Mapping between iOS UI components and Android UI components.**

<table>
<thead>
<tr>
<th>iOS UI component</th>
<th>Android UI component</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIActivityIndicatorView</td>
<td>Spinner</td>
</tr>
<tr>
<td>UIAlertView</td>
<td>AlertDialog</td>
</tr>
<tr>
<td>UIBarButtonItem</td>
<td>ActionBar</td>
</tr>
<tr>
<td>UIItem</td>
<td>MenuItem</td>
</tr>
<tr>
<td>UIButton</td>
<td>Button</td>
</tr>
<tr>
<td>UIColor</td>
<td>Color</td>
</tr>
<tr>
<td>UIDatePicker</td>
<td>DatePicker</td>
</tr>
<tr>
<td>UIButtonEvent</td>
<td>Event</td>
</tr>
<tr>
<td>UIFont</td>
<td>Font</td>
</tr>
<tr>
<td>UIImage</td>
<td>Image</td>
</tr>
<tr>
<td>UIImage</td>
<td>ImageView</td>
</tr>
<tr>
<td>.getOwnPropertyView</td>
<td>KeyboardView</td>
</tr>
<tr>
<td>UILabel</td>
<td>TextView</td>
</tr>
<tr>
<td>UIMenuItem</td>
<td>MenuItem</td>
</tr>
<tr>
<td>UNavigationBar</td>
<td>ActionBar</td>
</tr>
<tr>
<td>UIPageViewController</td>
<td>ViewPager</td>
</tr>
<tr>
<td>UIPickerView</td>
<td>DatePicker</td>
</tr>
<tr>
<td>UIProgressView</td>
<td>ProgressBar</td>
</tr>
<tr>
<td>UIScrollView</td>
<td>ScrollView</td>
</tr>
<tr>
<td>UISearchBar</td>
<td>SearchView</td>
</tr>
<tr>
<td>UISegmentedControl</td>
<td>RadioGroup</td>
</tr>
<tr>
<td>UISlider</td>
<td>SeekBar</td>
</tr>
<tr>
<td>UISplitViewController</td>
<td>RelativeLayout</td>
</tr>
<tr>
<td>UISlider</td>
<td>Button</td>
</tr>
<tr>
<td>UISwitch</td>
<td>ToggleButton</td>
</tr>
<tr>
<td>UITabBar</td>
<td>TabHost</td>
</tr>
<tr>
<td>UITabBarItem</td>
<td>TabWidget</td>
</tr>
<tr>
<td>UITableView</td>
<td>ListView</td>
</tr>
<tr>
<td>UITableViewCell</td>
<td>TableRow</td>
</tr>
<tr>
<td>UTextField</td>
<td>EditText</td>
</tr>
<tr>
<td>UTextPosition</td>
<td>FieldPosition</td>
</tr>
<tr>
<td>UTextView</td>
<td>TextView</td>
</tr>
<tr>
<td>UIToolbar</td>
<td>ActionBar</td>
</tr>
<tr>
<td>UIView</td>
<td>Layout</td>
</tr>
<tr>
<td>UIWebView</td>
<td>WebView</td>
</tr>
<tr>
<td>UIWindow</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Table 5: UI aspects mapping between iOS and Android.**

<table>
<thead>
<tr>
<th>UI aspect</th>
<th>iOS</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text alignment</td>
<td>Centered</td>
<td>Left</td>
</tr>
<tr>
<td>Navigation bar</td>
<td>Tab bar menu with centered items</td>
<td>Bottom navigation located at the bottom</td>
</tr>
<tr>
<td>Font family</td>
<td>San Francisco and Helvetica Neue</td>
<td>Roboto</td>
</tr>
<tr>
<td>Button styles</td>
<td>Flat button with shadows</td>
<td>Floating action buttons with text</td>
</tr>
<tr>
<td>Icons</td>
<td>Thin line icons</td>
<td>Thick stroke icons</td>
</tr>
<tr>
<td>Controls</td>
<td>Tabs or buttons</td>
<td>Simple underline</td>
</tr>
<tr>
<td>List</td>
<td>With arrow icons</td>
<td>No arrow icons on the right side</td>
</tr>
</tbody>
</table>

**Table 6: iPhone device screen sizes.**

<table>
<thead>
<tr>
<th>Device</th>
<th>Portrait dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone, iPhone 4, 4s</td>
<td>320 dp × 480 dp</td>
</tr>
<tr>
<td>iPhone 5, 5s, 5c, SE</td>
<td>320 dp × 688 dp</td>
</tr>
<tr>
<td>iPhone 6, 6s, 7, 8</td>
<td>375 dp × 667 dp</td>
</tr>
<tr>
<td>iPhone 6+, 6s+, 7+, 8+</td>
<td>414 dp × 736 dp</td>
</tr>
<tr>
<td>iPhone X, xs</td>
<td>375 dp × 812 dp</td>
</tr>
<tr>
<td>Autolayout</td>
<td>600 dp × 600 dp</td>
</tr>
</tbody>
</table>

**Table 7: Android device screen sizes.**

<table>
<thead>
<tr>
<th>Device</th>
<th>Portrait dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Pixel</td>
<td>411 dp × 731 dp</td>
</tr>
<tr>
<td>HTC One M9</td>
<td>360 dp × 640 dp</td>
</tr>
<tr>
<td>LG G3</td>
<td>480 dp × 853 dp</td>
</tr>
<tr>
<td>Moto X</td>
<td>360 dp × 640 dp</td>
</tr>
<tr>
<td>Nexus 5</td>
<td>360 dp × 640 dp</td>
</tr>
<tr>
<td>Nexus 7</td>
<td>600 dp × 960 dp</td>
</tr>
<tr>
<td>Samsung Galaxy S5</td>
<td>360 dp × 640 dp</td>
</tr>
<tr>
<td>Samsung Galaxy S8</td>
<td>360 dp × 740 dp</td>
</tr>
<tr>
<td>Sony Xperia Z3</td>
<td>360 dp × 668 dp</td>
</tr>
</tbody>
</table>

**Table 8: Android small width metrics.**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Smallest width</th>
</tr>
</thead>
<tbody>
<tr>
<td>320 dp × 569 dp</td>
<td>sw320dp</td>
</tr>
<tr>
<td>360 dp × 640 dp</td>
<td>sw360dp</td>
</tr>
<tr>
<td>384 dp × 640 dp</td>
<td>sw384dp</td>
</tr>
<tr>
<td>411 dp × 731 dp</td>
<td>sw411dp</td>
</tr>
</tbody>
</table>

(vi) Scalability: this refers to how well the MDRE tool works when the size of the system being studied increases

(vii) Validation: has the approach been validated by industrial case studies, or was it just in an early stage?
4.5.2. Application Domain. The scope of the proposed approach is specifically designed for reverse-engineering mobile platforms field. In this work, the authors focused on the generation of functional specs and Android UI. These artifacts are useful for developers to bootstrap the full porting from iOS to Android. Nevertheless, the approach itself is also valid for other purposes. The strong point of the current approach lies in representing the mobile application in the form of PSM models. In turn, any information or data from the application is preserved for later processing. There are others use cases in which the current approach can be exploited, for example:

(i) Upgrading iOS apps written in Objective-C to the latest Swift language version
(ii) Refactoring some APIs towards more compliant and stable ones
(iii) Reverse engineering of the iOS UI part to Microsoft Xamarin UI or React Native
(iv) Reverse engineering from Android to iOS

4.5.3. Extensibility. The proposed approach guarantees a separation between the models and the actions of each stage of the reverse-engineering process. Indeed, the implementation
of the current approach includes facilities to expand meta-
models or add new ones and the abilities to define new
transformation rules as well as to define new generator’s
templates.

4.5.4. Automation. The current approach is totally auto-
mated, with regard to the scope of generating the functional
specs and the UI part. However, it is plausible that a number
of limitations may be raised when it comes to the mapping
between platform SDKs.

4.5.5. Full/Partial Coverage. In the proposed approach,
metamodels are described in order to cover all the views of
the source and target mobile platforms.

Figure 11: Overview of specific Xpand templates designed for the Android UI project.

Figure 12: Overview of the generated functional spec document.
5. Tool Implementation: iSpecSnapshot

5.1. Description. This section describes iSpecSnapshot tool as a reverse-engineering tool, which implements the proposed MDRE approach. As stated in Introduction, the main aim of this tool is to produce spec documentation for a given native iOS mobile app and reverse-engineer the Android UI part. iSpecSnapshot is a model-based toolbox, an experimental tool, developed by MyAppConverter team as a part of MyAppConverter SAAS platform. Figure 13 shows the technical architecture of iSpecSnapshot.

iSpecSnapshot is structured as a set of loosely coupled collaborating services. Each service is registered with a so-called service registry. The registration process is periodically refreshed following a heartbeat mechanism. The service’s logic implements narrowly related subservices. It runs as a unique process and communicates through a distributed streaming platform to serve each MDRE stage. Note that all services are tightly communicating with the underlying repository cluster. The cluster includes the following repositories: metamodel repository, knowledge base, and template repository.

5.1.1. Metamodel Repository. It includes in particular all the metamodels, which are specific to the mobile platforms. Metamodels are classified by platform type and by domain such as infrastructure, technological, and platform-specific metamodel. For example, for the iOS platform, the metamodels are listed as follows:

(i) Infrastructure: Xcode Project metamodel
(ii) Technological: Objective-C Metamodel, Swift Metamodel
(iii) Platform-specific metamodels: storyboard metamodel, plist metamodel, and xcdata metamodel

5.1.2. Knowledge Base KB. It houses the mapping between the SDKs APIs of the two platforms (iOS and Android). The KB is structured as a graph database, each node represents an entity (header/java package, class, interface, method, parameter, access kind, etc.), and each relationship represents how two nodes are associated. Using Graph databases have many key advantages, such as performance, flexibility, and agility.

5.1.3. Template Repository. It includes all kind of templates that are used in the code-generation phase. It is also structured by domains: infrastructure, technological, documentation, and applicative. The following classification is considered:

(i) Infrastructure: it describes the Android studio project template
(ii) Technological: it describes the Java AST template
   (a) Declaration block: class, interface, method, etc.
   (b) Statement block: if, For, While, etc.
(iii) Applicative: it describes the UI components (TextView, Button, List, Image, TextView, Switch, etc.)
(iv) Documentation: it describes all sections of the functional spec document: Cover page, UML diagram page, Activity diagram section, Coverage report, etc.

From a usability view, the user experience of the tool is ergonomically smart and intuitively simple (Figure 14). Indeed, to get started with a porting session, the end user should proceed as follows: first create an account, then upload the source code of the iOS app, start the porting process, visualize the UI preview, and analyze the generated outputs (SFD document and Android UI project).
From a logical view, iSpecSnapshot provides four services:

(i) Parsers allow to automatically generating PSM models from the iOS mobile app source code. The PSM models correspond to Xcode project structure, programming language (C, C++, Objective-C, and Swift), xib file, plist file, strings file, etc.

(ii) Extractors analyze the PSM models, which are produced by the parsers, in order to extract semantic information for building the PIM pivot model (semantic mobile model). The PIM model typically identify specific information on the iOS app: name, icon, build, device orientation, linked frameworks, screen workflow, lifecycle behavior, navigations, transition, UI component properties, etc.

(iii) Transformers explore the PIM model and launch a recursive transformation process in order to build the documentation PSM model and the Android PSM model. SFD’s PSM model describes in a reliable manner the whole structure and content of the functional specification documentation: Cover page, summary page, screen page, framework dependencies and mapping, entity models, persistent models, activity diagram, etc. The Android app PSM model describes the structure of the target Android UI project.

(iv) Generators move around the resulting SFD PSM and Android PSM models and load the corresponding templates from the template repository in order to generate the SFD document and the Android UI skeleton.

5.2. Limitations. The authors sound a note that their approach supports currently the reverse engineering of iOS applications that have the UI part statically defined in the storyboard or Xib files. Indeed, the majority of the UI components of the UIKit framework are covered (UIView, UILabel, UIButton, UITextField, UITextView, UISlider, UISegmentedControl, UISwitch, UITableView, etc.), except some restrictions compared to matching semantic properties with their Android correspondents. However, the current research is in progress to support UIKit applications whose UI part was dynamically programmed or customized in Swift or Objective-C. The major challenge in this task is how to recognize code blocks, where developer uses the instructions for building, instantiating, initializing, and invoking UI components inside the controller’s AST tree.

6. Evaluation

The implemented tool is evaluated according to two aspects: evaluation as a tool that meets software development standards and evaluation as an effective mobile porting tool. The evaluation of the tool with regard to software developments standard includes two testing phases: alpha testing and private beta testing. In lockstep with the MyAppConverter team, the authors have conducted alpha testing stage to avoid any bugs or issues and to guarantee that the tool meets perfectly the established requirements. At the end of this stage, private beta testing was conducted with a close group of developers selected from the interview sessions. The aim of the private beta testing is to collect feedbacks on how to improve the overall user experience of the tool, as well as fixing some of the issues found in the generated outputs.

To evaluate the proposed tool as an effective mobile porting tool for mobile developers, the authors used an experimental procedure within the selected group of beta-test users. The objective of this process is to ensure that the resulting outputs meet the expectations of the developers in terms of functional specs details extracted from the iOS source app and the UI quality of the Android project ported from iOS. Therefore, they try to answer to the fundamental problematic of the current research: RQ: Does iSpecSnapshot efficiently help mobile developers to speed up the mobile porting process?

To assess an efficient evaluation, the authors will carefully focus on apps that are built using UIKit framework (iOS UIKit Framework: https://developer.apple.com/documentation/uikit). And obviously, they will be particularly interested in feedbacks from two categories of users: Android developers and functional analysts. The authors also did look for other tools to compare with iSpecSnapshot’s features. But so far, they could not find any tool whose functional scope can be compared to iSpecSnapshot tool (generating functional spec and Android UI part). Nevertheless, there are commercial tools.
whose technical documentation is quite poor and which have the vocation of binary decompiler or interactive disassembler or even debugger, such as Veracode iRet (iRet Toolkit: https://www.veracode.com/iret-ios-reverse-engineering-toolkit-veracode), IDA (DA Pro: https://www.hex-rays.com/products/ida/), and Hopper (Hooper: https://www.hopperapp.com).

6.1. Participants. The authors have recruited 13 participants, selected from the interview sessions. The participants are divided into two categories. The first is composed of 10 Android developers with minimal skills of the iOS platform. The second involves 3 mobile functional analysts, who are accustomed to produce mobile functional spec document. The majority of the participants either hold positions at startups or are freelancers who are self-employed. All the participants were ensured about their privacy and data before their participation in this study. The data was solely used for research purposes and will not be shared with anyone under any circumstances.

6.2. Experimental Objects. The authors have carried out experiments on the beta-test user's apps. As was aforementioned, they have limited the experimental objects to 13 iOS apps, which are submitted by the selected participants. The choice of participants' apps is determined by 3 criteria. First, the application must be compatible at least with the Xcode7.x version. Secondly, the UI part should be designed according to the storyboard/xib files, and finally, the usage of the UIKit components should exceed 70% of all the used UI components (customized UI components will not be covered in the current scope of the tool). To preserve the confidentiality of apps supplied by participants, the authors have identified the apps by the identifier APP\textsubscript{n} (1 ≤ n ≤ 13) and participant by the identifier PAR\textsubscript{m} (1 ≤ m ≤ 13).

Table 9 shows some characteristics of each application: number of resources who has developed the iOS app, global development cost (man-day), workload of the UI part (man-day), number of screens, total number of the UI component, and UI complexity. As it emerges from the data reported in Table 9, the selected apps are sufficiently diverse, since they represent different functional scopes and have variable metrics in terms of workload and size.

6.3. Metrics Definition. Since, in the present work, the authors focused on reverse engineering of the functional spec and UI porting, they merely point out that, in their experiments, they decide to evaluate the effectiveness of iSpecSnapshot with regard to the accuracy of the functional spec formulation, coverage of framework mapping, coverage of the UI components, and cost margin of the generated Android UI part. Therefore, the following metrics are considered:

(i) Functional spec accuracy percentage (FSA%) represents the percentage of consistency/completeness of the information represented in the FSD document, including the persistence model (PM%), the business class model (BM%), and the activity diagram (AM%). It can be measured according to the following formula:

\[
\text{PM\%} = \frac{\text{number of detected items for the persistence model}}{\text{total number of the persistence model items}} \times 100, \\
\text{BM\%} = \frac{\text{number of detected items of the business model}}{\text{total number of the business model items}} \times 100, \\
\text{AM\%} = \frac{\text{number of detected screen transition}}{\text{total number of screen transition}} \times 100, \\
\text{FSA\%} = \text{mean}(\text{PM\%}, \text{BM\%}, \text{AM\%}).
\]  

(ii) Coverage of framework mapping percentage (CFM \%) defines the percentage of the mapped iPhone SDK elements used in the iOS source code, with their correspondent in the Android SDK.

\[
\text{CFM\%} = \frac{\text{number of mapped iPhone SDK elements}}{\text{number of iPhone SDK elements used in the source code}} \times 100.
\]  

(iii) Coverage of UI component percentage (CUI\%) defines the percentage of the generated Android UI component against the overall of the UI components of the iOS App.
CUI\% = \sum_{n=1}^{sn} \left(\frac{\text{number of generated UI component per each screen}}{\text{total number of UI component per each screen}}\right) \times 100, \quad \text{sn = screen numbers.} \quad (4)

(iv) Cost margin of the generated Android UI part percentage (CMUI\%) defines the cost margin involved by the generated Android UI part against a from scratch development task.

\[
\text{CMUI\%} = 100 \times \left(1 - \frac{\text{EffAutoGenUI} + \text{EffCustomUI}}{\text{EstimEffAndroidUIMan}}\right),
\]

\text{EffAutoGenUI} = \text{effort of the automatic generation of UI part,}

\text{EffCustomUI} = \text{effort of developing customized UI component,}

\text{EstimEffAndroidUIMan} = \text{estimated effort for developing the Android UI part manually.} \quad (5)

6.4. Experimental Design. The experimental methodology which is performed for carrying out the study consists of three sequential steps: source app analysis, outputs comparison, and metrics analysis.

In the source app analysis step, one researcher and a senior iOS developer, from the MyAppConverter team, had the task of preparing an analytical report for each iOS application from the list of selected objects. The report includes the following information: number of screens, number of standard UI components per screen, number of customized UI components, UI complexity, transition between screens (defined using storyboard connections or managed by code), and referenced pods or third libraries. The analysis task consists also of revisiting the iOS PSM specific to each app. The goal is to ensure that the model resulting from the model discovery phase does not contain errors, which may lead to false results for the other stages of the MDRE process.

In the outputs comparison step, the authors divided the selected subjects into 2 groups: G-UI includes Android developers who have the task of reviewing the Android UI project generated by iSpecSnapshot tool and compare it to the UI part of their iOS app. The second G-SPEC group had the task of evaluating the content of the functional spec document generated for all the apps. The analysis of the participants is even more important, since it gives insight a critical view of the end customer in relation with the results generated by the evaluated tool. However, the authors believe that limiting themselves to their own judgment, in terms of results, can bias the overall evaluation of the tool. To assess the generated Android UI project, they instruct the participants to import the generated Android UI project into Android Studio IDE and they recommend using the 3.1.x version. The participants are invited to use the Xcode Interface Builder IDE 9.x to compare the iOS UI part with the

<table>
<thead>
<tr>
<th>iOS app ID</th>
<th>Owner</th>
<th>Developed by</th>
<th>Global dev. cost (man-day)</th>
<th>UI part workload (man-day)</th>
<th>No. of screen</th>
<th>No. of UI comp.</th>
<th>UI complexity</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP1</td>
<td>PAR1</td>
<td>2</td>
<td>30</td>
<td>6</td>
<td>18</td>
<td>158</td>
<td>Medium</td>
<td>Education</td>
</tr>
<tr>
<td>APP2</td>
<td>PAR2</td>
<td>2</td>
<td>68</td>
<td>12</td>
<td>42</td>
<td>347</td>
<td>Basic</td>
<td>Gaming</td>
</tr>
<tr>
<td>APP3</td>
<td>PAR3</td>
<td>1</td>
<td>43</td>
<td>9</td>
<td>19</td>
<td>144</td>
<td>Medium</td>
<td>Communication</td>
</tr>
<tr>
<td>APP4</td>
<td>PAR4</td>
<td>3</td>
<td>63</td>
<td>15</td>
<td>38</td>
<td>290</td>
<td>Complex</td>
<td>Social Team collaboration</td>
</tr>
<tr>
<td>APP5</td>
<td>PAR5</td>
<td>3</td>
<td>96</td>
<td>20</td>
<td>49</td>
<td>370</td>
<td>Complex</td>
<td>Health</td>
</tr>
<tr>
<td>APP6</td>
<td>PAR6</td>
<td>3</td>
<td>75</td>
<td>22</td>
<td>43</td>
<td>314</td>
<td>Medium</td>
<td>Networking</td>
</tr>
<tr>
<td>APP7</td>
<td>PAR7</td>
<td>2</td>
<td>54</td>
<td>11</td>
<td>23</td>
<td>130</td>
<td>Complex</td>
<td>Adverting</td>
</tr>
<tr>
<td>APP8</td>
<td>PAR8</td>
<td>1</td>
<td>15</td>
<td>3</td>
<td>9</td>
<td>53</td>
<td>Basic</td>
<td>Team collaboration</td>
</tr>
<tr>
<td>APP9</td>
<td>PAR9</td>
<td>2</td>
<td>24</td>
<td>5</td>
<td>25</td>
<td>211</td>
<td>Basic</td>
<td>Medicine</td>
</tr>
<tr>
<td>APP10</td>
<td>PAR10</td>
<td>2</td>
<td>20</td>
<td>4</td>
<td>14</td>
<td>107</td>
<td>Medium</td>
<td>Events</td>
</tr>
<tr>
<td>APP11</td>
<td>PAR11</td>
<td>1</td>
<td>30</td>
<td>5</td>
<td>13</td>
<td>85</td>
<td>Medium</td>
<td>Gaming</td>
</tr>
<tr>
<td>APP12</td>
<td>PAR12</td>
<td>2</td>
<td>64</td>
<td>21</td>
<td>27</td>
<td>167</td>
<td>Complex</td>
<td>Social</td>
</tr>
<tr>
<td>APP13</td>
<td>PAR13</td>
<td>1</td>
<td>20</td>
<td>4</td>
<td>13</td>
<td>112</td>
<td>Basic</td>
<td>Medicine</td>
</tr>
</tbody>
</table>

Table 9: Metrics of the experimental objects: number of resources who has developed the iOS app, global development cost (man-day), workload of the UI part (man-day), number of screens, total number of the UI component, and UI complexity.
generated Android UI. It should be noted that the generated android xml layout is identified by the pattern activity_ <id of the iOS screen>.xml. To better help the members of each group in their analysis, the authors have configured the Trello (Trello: https://www.trello.com) agile tool. Trello is a collaboration tool that organizes projects into boards. They set up Trello boards for each app; then, they sent out email invitations to invite each participant to join the corresponding board. Each participant is brought to report any remark or bug as "cards" in the "issues backlog" list. It is also interesting to specify the severity of each issue using "labels" decorators: "critical", "major", "medium," and "low". The bugs reported by each participant are supported by the researchers’ team; they are then moved to the "In progress" list of the Trello board. The qualification of the bugs will point out which phase of the MDRE process will require an adjustment to correct the bug. Once the bug is corrected, the card associated with the bug is then moved to the "Done" list. This process is repeated for each application until all the bugs are exhausted. Each participant is asked to renew a new iteration to make sure that all these remarks are taken care of.

In the data collection and analysis step, the authors analyzed the reports produced by each participant. At the end of the last iteration, each participant is brought to fill in his own Trello board the following metrics: the execution time (to report by the G-UI and the G-Spec groups), the percentage of detected elements from persistence model (reserved for the G-Spec group), the percentage of detected elements from business model (reserved for the G-Spec group), the percentage of mapped iPhone SDK vs. Android SDK (reserved for the G-Spec group), the percentage of detected screen transitions (reserved for the G-Spec group), the percentage of mapped UIKit components does not exceed few minutes on average 117 sec, for the current case study (Table 10). According to the current case studies, the authors have estimated the average effort of implementing Android UI part from scratch (Figure 17). However, the execution time displayed by iSpecSnapshot in order to automatically generate UIKit components does not exceed few minutes on average

$$\text{CMUI} = 100 \times \left(1 - \frac{\text{EstimEffAndroidUIMan} + (1/4)}{\text{EffCustomUI}}\right).$$

Thus, the gain in development effort of the UI Android part using iSpecSnapshot exceeds the threshold of 77% (Figure 18).

6.5. Experimental Results. Table 10 reports the resulting values of all the metrics measured for the selected apps. The results are also represented as graphs (Figures 15–17) to more appropriately analyze the FSA%, CFM%, and CMUI% metrics.

Based on the evaluation analysis of functional analysts, iSpecSnapshot visibly identifies the entity model, the business class model, and the activity diagram. Indeed, for the 13 apps, the average of FSA% is of 69.82%, which is quite encouraging (Figure 15).

The authors observe that iSpecSnapshot has clearly managed to identify the persistence model (PM% = 100%), for apps that use the xcdatamodel as a format to describe the persistence model.

Regarding the apps that have a BM% < 100 score (for example APP7, 47%), this is due to the fact that the business classes are coded in other languages other than Objective-C (the beta version of the parsing engine recognizes just the Objective-C language).

As for the apps that have a score AM% < 100 (for example APP4, 42%), this is justified by the fact that the flow of navigations between screens is partly programmed in the view controllers (the beta version of iSpecSnapshot is limited to connections of navigations defined in the storyboard).

Regarding CFM% of the coverage of mapped frameworks from iOS to Android, the knowledge base KB of iSpecSnapshot shows an acceptable average of 63% for the considered apps in this evaluation (Figure 16). Indeed, most of the studied apps use frameworks from the iPhone SDK that are not yet supported in the KB; in addition, there are many apps (App i, i ∈ [1, 4, 5, 6, 7, 12]) that use third libraries or specific pods.

For apps that are categorized under “UI complex” category, the estimated effort to implement specific UI components (components that are not part of the UIKit framework) typically represents a ratio between 20% and 30% of the estimated total charge for implementing Android UI part from scratch (Figure 17). However, the execution time displayed by iSpecSnapshot in order to automatically generate UIKit components does not exceed few minutes on average

$$\text{CMUI} = 100 \times \left(1 - \frac{\text{EstimEffAndroidUIMan} + (1/4)}{\text{EffCustomUI}}\right).$$

Thus, the gain in development effort of the UI Android part using iSpecSnapshot exceeds the threshold of 77% (Figure 18).

6.6. Study Conclusion. According to these experimental results, the authors could answer the main research question of their study concluding that iSpecSnapshot can greatly help Android developers in their tasks of porting iOS applications. On one hand, the tool draws up a report that gives visibility on the use of iOS frameworks and their Android mapping. On the other hand, the tool fills the lack of functional documentation by bringing a functional understanding of the iOS application in terms of persistence and business models. In addition, it allows tracing the flow of navigation between screens and describing in detail the components of each screen. Finally, it speeds up the development time and bootstrap the porting project, while sparing the Android developer of the effort to implement the Android UI part from scratch.

6.7. Threats to Validity. In this section, the authors discuss the validity threats in conducting this experimental study [39].

6.7.1. Internal Validity. The evaluation of the accuracy of the FSD document for the 13 apps by the group of 3 functional analysts may present a possible threat to the internal validity.
To avoid this threat, the authors have selected 3 candidates based on their skills, regarding the iOS platform, in order to be able to compare the generated results to the source code of the iOS app. The authors also have conducted a double check with the group of Android developers who are a priori the owners of iOS apps.

Table 10: Average metric values of the selected apps performed by iSpecSnapshot.

<table>
<thead>
<tr>
<th>iOS app ID</th>
<th>PM %</th>
<th>BM %</th>
<th>AM %</th>
<th>FSA %</th>
<th>CFM %</th>
<th>CUI %</th>
<th>Eff auto gen. UI (sec)</th>
<th>Eff custom UI (man-day)</th>
<th>Estim Eff Android UI Man (man-day)</th>
<th>CMU %</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP1</td>
<td>100</td>
<td>65</td>
<td>62</td>
<td>75.67</td>
<td>59</td>
<td>68</td>
<td>96</td>
<td>1.58</td>
<td>6</td>
<td>69.54</td>
</tr>
<tr>
<td>APP2</td>
<td>73</td>
<td>92</td>
<td>55.00</td>
<td>71</td>
<td>94</td>
<td>209</td>
<td>0.65</td>
<td>12</td>
<td>92.50</td>
<td></td>
</tr>
<tr>
<td>APP3</td>
<td>100</td>
<td>90</td>
<td>67</td>
<td>85.67</td>
<td>63</td>
<td>60</td>
<td>87</td>
<td>1.78</td>
<td>9</td>
<td>77.25</td>
</tr>
<tr>
<td>APP4</td>
<td>98</td>
<td>42</td>
<td>46.67</td>
<td>53</td>
<td>41</td>
<td>175</td>
<td>5.34</td>
<td>15</td>
<td>62.74</td>
<td></td>
</tr>
<tr>
<td>APP5</td>
<td>100</td>
<td>53</td>
<td>51</td>
<td>68.00</td>
<td>48</td>
<td>43.5</td>
<td>223</td>
<td>6.52</td>
<td>20</td>
<td>66.14</td>
</tr>
<tr>
<td>APP6</td>
<td>100</td>
<td>84</td>
<td>65</td>
<td>83.00</td>
<td>58</td>
<td>65</td>
<td>190</td>
<td>3.43</td>
<td>22</td>
<td>83.28</td>
</tr>
<tr>
<td>APP7</td>
<td>100</td>
<td>47</td>
<td>49</td>
<td>65.33</td>
<td>54</td>
<td>44</td>
<td>79</td>
<td>2.27</td>
<td>11</td>
<td>77.08</td>
</tr>
<tr>
<td>APP8</td>
<td>100</td>
<td>100</td>
<td>66.67</td>
<td>78</td>
<td>89</td>
<td>32</td>
<td>0.18</td>
<td>3</td>
<td>85.60</td>
<td></td>
</tr>
<tr>
<td>APP9</td>
<td>67</td>
<td>100</td>
<td>55.67</td>
<td>83</td>
<td>93</td>
<td>128</td>
<td>0.46</td>
<td>5</td>
<td>85.78</td>
<td></td>
</tr>
<tr>
<td>APP10</td>
<td>100</td>
<td>69</td>
<td>89.67</td>
<td>64</td>
<td>58</td>
<td>71</td>
<td>1.40</td>
<td>4</td>
<td>58.70</td>
<td></td>
</tr>
<tr>
<td>App11</td>
<td>100</td>
<td>51</td>
<td>70</td>
<td>73.67</td>
<td>67</td>
<td>56</td>
<td>52</td>
<td>1.17</td>
<td>5</td>
<td>71.66</td>
</tr>
<tr>
<td>App12</td>
<td>100</td>
<td>74</td>
<td>54</td>
<td>76.00</td>
<td>49</td>
<td>52</td>
<td>102</td>
<td>2.80</td>
<td>21</td>
<td>85.47</td>
</tr>
<tr>
<td>App13</td>
<td>100</td>
<td>100</td>
<td>66.67</td>
<td>81</td>
<td>92</td>
<td>69</td>
<td>0.28</td>
<td>4</td>
<td>86.76</td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td>69.82</td>
<td>63.69</td>
<td>65.81</td>
<td>116.38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>77.12</td>
<td></td>
</tr>
</tbody>
</table>
6.7.2. **External Validity.** In this study, a possible threat to the external validity could have been the selection of the object apps. The goal of the first step of the evaluation process is to mitigate this threat. During this preliminary analysis, the authors have selected a set of different applications to cover the entire range of UIKit components. They also have been interested in applications that deal with various functional domains (social, medicine, networking, etc.). In order to extend the validity of their results, other case studies were considered during the public beta phase of the tool.

7. **Conclusion and Future Work**

To meet the growing demands of mobile app porting, new software engineering techniques and reverse-engineering tools adapted to the mobile platform remain essential in order to help developers to better understand, analyze, and maintain mobile applications.

In this paper, the authors have presented iSpecSnapshot, a model-driven reverse-engineering tool, which aims to generate automatically the functional specification documentation of iOS mobile apps and bootstrap the
corresponding Android UI skeleton. They have underlined their MDRE approach and outlined the key reverse-engineering steps: model discovery, semantic extraction, model transformation, and code generation. Further, they have exposed the technical architecture of iSpecSnapshot as a cloud service.

In the meantime, the authors have revised some preliminary works in mobile the reverse-engineering field. The findings of their evaluation, conducted on iOS native apps, have ensured that iSpecSnapshot is a useful solution for detecting key information for the iOS to Android mobile app porting process, e.g., framework dependencies, rate of SDK platforms mapping, activity diagram, business model, and screen components (UI elements, events and actions).

7.1. Future Work. There are some research lines in which the proposed approach can be improved and extended. The immediate task would be to improve the activity diagram to support dynamic screen transitions. This could eventually be done by readjusting the semantic extraction step, in order to detect the dynamic navigation that is described towards the source code of event actions. This leads certainly to further enrichment of the semantic repository, based on the best practices of iOS mobile development.

Other perspectives worthwhile pursuing would be to enhance the mapping coverage rate between mobile SDK frameworks. In fact, the authors are looking forward to examine the possible mapping between iPhone SDK and the plethora of Android open-sourced libraries. That point they hope to be valuable for a full-porting process.

Furthermore, the prospect of being able to port from iOS to Android serves as a continuous incentive for other reverse-engineering directions. The authors are about to prototype a reverse-engineering POC (Proof Of Concept) from UI iOS to Flutter UI (Flutter UI: https://flutter.io). They intend to follow the same MDRE approach that is presented in this paper in order to migrate the iOS UI to Google’s portable UI Toolkit, which is known by their fast rendering and expressive UI.

Appendix

A. Model-Driven Engineering (MDE)

MDE [40] is a growing concept in software engineering. It aims to improve software development by raising the level of abstraction in system specification and increase automation of implementation, maintenance, and testing [41, 42]. The promoted idea by MDE is to use models at different levels of abstraction for developing systems. The MDE approach is widely used in the context of forward engineering [43], through appropriate model to code M2C tools specifically designed to convert model to code.

The object management group (OMG) describes models as “a description or specification of a system and its environment for a certain purpose” [44]. The model is related to the system by an explicit or implicit mapping. It may represent the software, the hardware, the environment, or other domain-specific aspects of the system. The OMG’s framework, i.e., MDA (model-driven architecture) [45], distinguishes different kinds of models (see Figure 19):

1. Computation independent model (CIM)
   Created by business analyst to describe business viewpoint

2. Platform independent model (PIM)
   Created by developer or tester to implement system

3. Platform specific model (PSM)
   Created by architect/designer to describe system architecture

4. Implementation-specific model (ISM)
   Created by coder to develop code

Figure 19: Key models of the MDA paradigm: CIM, PIM, and PSM.
A PSM is a view of a system from the platform-specific viewpoint. It combines the specifications in the PIM with the details that specify how the system uses a particular type of platform. The PSM includes specific elements representing the target platform [46].

An ISM is a model which all details of implementation of the target system are specified [47].

Metamodel defines the abstract syntax of a modeling language and, as such, is a special kind of model. It can be described as the specification of the set of all possible models expressed in that modeling language. MDA is based on the use of a language to write metamodels called the meta object facility (MOF) [48]. MOF guarantees that the models can be stored in a MOF-compliant repository, parsed, transformed, rendered into different formats including XMI (XML Metadata Interchange), transported across a network, and used to generate application code.

Model transformation is a very important task in any model-driven approach. Indeed, the transformations ensure the operations of translating of one or more models from a given level of abstraction to one or more other models of the same level (horizontal transformation: PSM to PSM) or of a different level (vertical transformation: PIM to PSM). The OMG has standardized QVT (query view and transformation) as transformation language specifications (query/view/ transformation or QVT [34] and MOF2Text [49]). The QVT standard defines three languages for transformations: operational, relational and core languages. The relational language is the most relevant [50].

B. Model-Driven Reverse Engineering (MDRE)

As noted by Chikofsky and Cross [51], the term “reverse engineering” is generally understood to mean “reverse engineering is the process of analyzing a subject system, to identify the systems components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.”

In software engineering, the use of reverse-engineering paradigm has been considered in order to extract a wide range of models from existing systems. Theses models help to analyze and comprehend the system.

In literature, model-driven reverse engineering (MDRE) [52] has come to be used to refer to model-driven engineering (MDE) approaches and techniques, which are applicable to the reverse-engineering challenge.

A recent review of this topic in the literature [53] shows that MDRE has reached a peak in publication within the last few years. This is mostly due to the OMG’s effort to provide the ADM (architecture-driven modernization) task force [54] with the purpose of standardizing metamodels for reverse-engineering legacy systems.

ADM has set up many key standards [55] in order to support all tasks that are subject to ADM-based process. Subsequent standards [56] will treat many aspects such as analysis, visualization, refactoring, and transformation-related standards:

(i) KDM (knowledge discovery metamodel) defines a metamodel that describes a high-level view of the application behavior, structure, and data.

(ii) ASTM (abstract syntax tree metamodel) defines a metamodel that represents the procedural level of the application. ASTM is built upon KDM to ensure language metadata exchange. Moreover, it allows analysis tools to deal with metamodel instead of the specific language AST.

(iii) SMM (structured metrics metamodel) defines a metamodel that provides metrics from the KDM that can describe technical, functional, and architectural issues.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was partially supported by MyAppConverter Ltd. Warm thanks are due to some employees at MyAppConverter LTD, who generously shared with the authors their ideas and their enlightenment, which had greatly contributed to the originality of this paper.

References


Mobile Information Systems


