JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 37, 107-121 (2021)
DOI: 10.6688/JISE.202101_37(1).0008

Navigation Flow Modeling as a Basis for the Automatic Generation of Android APPs*

HSI-MIN CHEN[1], TSUNG-CHI LIN[1], LIEN-WU CHEN[1], BAO-AN NGUYEN[1]
AND YI-CHUNG CHEN[2]
[1]*Department of Information Engineering and Computer Science*
Feng Chia University
Taichung, 40724 Taiwan
[2]*Department of Industrial Engineering and Management*
National Yunlin University of Science and Technology
Yunlin, 64000 Taiwan
E-mail: {hmchen; M9802118; lwuchen}@mail.fcu.edu.tw;
baoanth@gmail.com; chenyich@yuntech.edu.tw

According to a marketing survey conducted in 2017, Android currently accounts for the largest market share of smartphone operating systems. However, demand for increased functionality from smartphone applications (apps) has greatly complicated program logic, data structure and operational flow. A lack of systematic schemes for the development of complex apps often hinders initial delivery and maintenance, compromises software quality, and leads to discrepancies between design specifications and implementations. In this study, we developed a novel scheme to facilitate the development of Android apps from the perspective of navigation flow (*i.e.*, a series of screens through which one navigates in order to perform a specific function related to a specific mobile app). The proposed model-driven architecture (MDA) makes it possible to automatically transform navigation flows (specified in wireframes) into Android project code within the context of the Android project structure. The objective of the proposed system is to enhance productivity in the development and maintenance of Android apps.

Keywords: mobile app development, Android, wireframe, navigation flow, MDA

1. INTRODUCTION

The world-renowned market research organization IDC Research reported that the sale of smartphones exceeded 344 million in the first quarter of 2017 [1]. At that time, the Android operating system [2] accounted for 85% of the global smartphone market, whereas Apple's iOS [3] accounted for only 14.7%.

However, demand for increased functionality from smartphone applications (apps) has greatly complicated program logic, data structure, and operational flow. A lack of systematic schemes for the development of complex apps often hinders delivery, compromises software quality, and leads to discrepancies between design specifications and implementation. Furthermore, app developers face the following challenges:

• *Tight development schedule*: At present, there are more than 3 million apps on Google Play. A Nine Hertz survey of 100 iOS, Android, and Web engineers revealed that 18

Received November 20, 2018; revised March 2, 2019; accepted June 4, 2019.
Communicated by Hung-Yu Kao.
* This research was supported in part by Ministry of Science and Technology, Taiwan, under Grants No. 107-2221-E-035-025-MY2.

weeks is the average time required for an app to move from initial development to launch [4]. This short development window has exacerbated the problem of implementing user requirements. Effective software development methods are required to enhance productivity in the development of new apps in an increasingly competitive market.

- *Inconsistencies between app design and implementation*: Mobile apps undergo more updates more frequently than do conventional desktop applications in response to user demands. Under these conditions, there is a constant gap between the intended design and its implementation, with the result that many apps are of low quality and are difficult to maintain.

We addressed the above challenges by examining the process of app development employed by software companies in this research. One common method is the drawing of wireframes to capture the user experience (UX) and to clarify the requirements of user interfaces [5]. Wireframes (drawn by hand or using rendering tools) are meant to unify the requirements of all stakeholders using easy-to-understand images. App wireframes can also be used to illustrate screen navigation flow (SNF) from one screen to another in the same manner that the user would follow when seeking to perform one of the app functions.

In this study, we developed a novel scheme aimed at facilitating the development of Android apps by screen navigation flow specified in the early design stage. We adopted a model-driven architecture (MDA) [6, 7] to enable the automatic generation of a corresponding code skeleton for a given navigation flow. The proposed scheme greatly reduces the time required to write code for app development. The ability to generate Android code directly from the wireframe design model largely overcomes the discrepancies commonly seen between the design of an app and its implementation. Furthermore, the addition of new functions and the modification of existing functions can easily be implemented within the design model to facilitate maintenance.

The remainder of this paper is organized as follows. Section 2 reviews related literature. Section 3 presents the proposed app development scheme. The experiment used to assess the efficacy of the scheme is outlined in Section 4. Conclusions are drawn in Section 5.

2. RELATED WORK

2.1 Android App Modeling

Several methods have been developed for the modeling of Android apps, including those based on static structures and dynamic behaviors. Unified modeling language (UML) [8] is the tool most widely used to model Android apps using a set of diagrams to capture the structure, behavior, and interactions of the system. Ko *et al.* [9] extended the notation and syntax of standard UML class diagrams by defining meta-models for static structural elements, the dynamic element lifecycle, and user interface components. These meta-model extensions make it easier for app designers to specify the requirements for app functions with greater precision. Parada *et al.* [10] employed UML class diagrams to provide a structural view of apps while using sequence diagrams to describe behavior. App designers can add app-specific classes to those pre-defined in the system to represent static relationships among classes and model dynamic interactions using sequence flows.

Several methods have been developed for the modeling of Android apps through the use of self-defined modeling languages. Yang *et al.* [11] defined window transition graphs (WTG) to describe series of graphical user interface (GUI) windows, relevant events, and callback functions. Specifically, WTG is used to model a stack of windows as well as changes and callbacks associated with the window stack. The modeling results can serve as inputs for the static analysis of Android apps. Most app modeling methods employ graphic representation to describe static structures and dynamic behaviors. Unlike text-based design specifications, the use of graphic representations for modeling can clarify for all stakeholders the overall design of the app from various perspectives. This approach also enables communication among developers dealing with widely disparate aspects of a given app.

2.2 MDA-Based Development of Android Apps

Over the last decade, MDA technologies have been widely applied in the development of software. MDA is generally used to generate source code from concept models through the application of various transforms. The highest abstract model is referred as to the platform-independent model (PIM), which is a system model that is independent of specific platforms/languages. PIMs can be transformed into platform-specific models (PSMs) through the application of clearly defined transformation rules. PSMs then undergo a further transformation into source code or text documents. MDA is crucial to the automatic generation of this kind of code.

Lachgar *et al.* [12] formulated a technology-neutral domain-specific language for use in modeling the user interface (UI) components of Android apps. The resulting PIMs are then transformed into various PSMs for WinPhone, iOS, and Web. Parada *et al.* [10] adopted a UML class diagram to provide structural views in conjunction with sequence diagrams to illustrate the behavior of apps. They then used GenCode [13] to generate Android-related code.

Unlike the studies mentioned above, our method employs wireframes as source models (obtained from UI/UX designers early in the design phase) to avoid the time and expense involved in drawing and learning a number of different models.

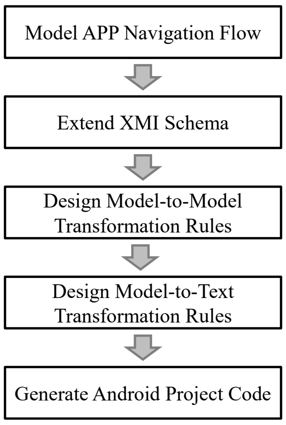

Model APP Navigation Flow
Extend XMI Schema
Design Model-to-Model Transformation Rules
Design Model-to-Text Transformation Rules
Generate Android Project Code

Fig. 1. MDA-based code generation process.

3. PROPOSED APPROACH

In this section, we outline the proposed approach to generating Android project code from wireframes. Fig. 1 depicts the steps involved in the code generation process, each of which is detailed below.

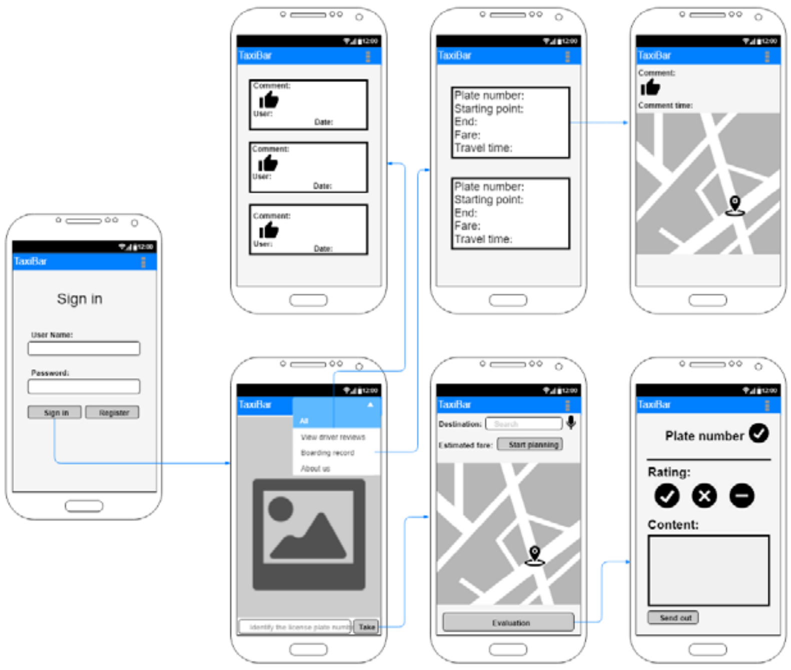


Fig. 2. Example of wireframe design.

3.1 Model Screen Navigation Flow

Unlike existing schemes, our approach involves the extraction of screen navigation flows from wireframes specifying the UI screens of apps and the navigation between them.

Fig. 2 presents an example of a wireframe design for the TaxiBar APP [14]. We can see that the app designer first designed the login screen, which describes the UI components and screen layouts as well as imposes a navigation flow from the login button (blue arrow) to the next screen. When a user clicks the login button and authentication is successful, the app navigates from the login screen to the main screen of the app. After confirming the design specifications for each screen, UI designers can submit them to subsequent developers.

Each sequence of switches from one screen to the next forms an SNF, which carries the user through the various steps involved in completing one function of the app. UML State Machine diagrams [15] are used to formalize these screens as well as the switches between them. A UML State Machine diagram comprises three elements: states, events, and transitions. When an event is initiated, one of the system states transitions to another state. State Machine diagrams are helpful in modeling the dynamic behaviors of systems.

Table 1. Semantic mapping between State Machine notations and app wireframes.

Notation	Semantics in State Machine	Semantics in app wireframes
●	Initial state	Enter an app
◉	Final state	Exit an app
▭	Simple state	An app screen
→	Transition	Switch between two screens

In this study, the modeling of SNFs using State Machine diagrams begins with the mappings of notation to semantics, as shown in Table 1. SNFs designed in wireframes can be translated into formal State Machine diagrams using these mapping rules. Initial and final states describe the start and end of an app. Each screen presented within wireframes represents a simple state. Based on the behavior depicted in the wireframes, the performance of any operation using the UI components of an app (*e.g.* tapping a button) causes a switch from one state to another. This switching behavior can be modeled using a transition (such as an arrow) linking two simple states. This allows app designers to associate additional information (*e.g.*, events, conditions, and actions) with a particular transition. Fig. 3 illustrates the State Machine model derived from the wireframe given in Fig. 2.

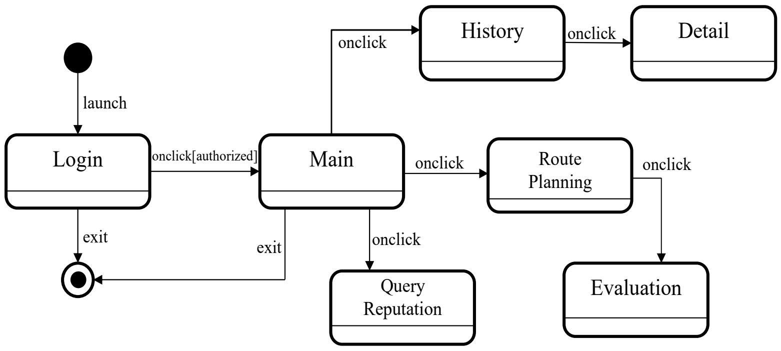


Fig. 3. State machine model derived from Fig. 2.

3.2 Extending XMI Schema

UML is a standard modeling language defined by the Object Management Group (OMG), and numerous software vendors have developed UML modeling tools to assist in the design of software systems. However, incompatibility among the tools has prevented the free exchange of designs produced using these tools. OMG defined XML Metadata Interchange (XMI) [16] to facilitate the free exchange of UML diagrams.

XMI also provides an extensible feature that allows users to define supplementary information and structures for existing meta-models. For example, modeling tools can be used to add coordinate information for graphic UI components to existing standard meta-models and save them in XMI files. This made it possible for us to add information required by automated app generation to the standard meta-model of the State Machine diagram.

When navigating among app screens, users can trigger an event on a UI component in order to switch to another screen. For example, if a user taps on the "login" button and is authorized, then the login screen switches to the main screen. However, the specifications of standard State Machine diagrams are not able to model UI components directly in diagrams. Thus, we use XMI to extend the meta-model of the State Machine diagram to which the supplementary information is attached. List 1 presents an example of an XMI extension for a State Machine diagram.

List 1. Example of XML extension for State Machine diagram

```
<transition xmi:id="trans_1" source="Login" target="Main">
  <eAnnotations xmi:id="ui_annotation_1" source="UI Component">
    <details xmi:id="ui_btn_login" key="Type" value="Button"/>
    <details xmi:id=" ui_btn_login _name" key="VariableName" value="loginBtn"/>
  </eAnnotations>
  <effect xmi:type="uml:FunctionBehavior" xmi:id="act_1" name="doTransition"/>
  <trigger xmi:id="trg_1" name="onClick" event="onClick_1"/>
</transition>
```

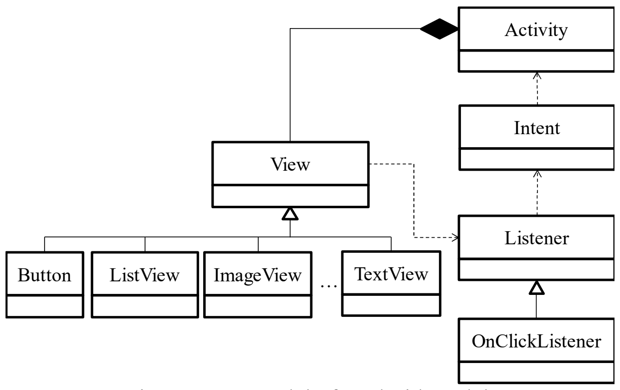


Fig. 4. Meta-model of Android model.

3.3 Model-to-Model Design Transformation Rules

After SNFs are modeled from wireframes into State Machine diagrams, the models are not translated directly into Android project code. This can be attributed to the fact that the structures in Android programs are updated less often than are the Android programming syntax and app programming interfaces (APIs). Thus, in this step, the State Machine diagrams are translated into PSMs based on definitions specified for the Android platform. Overall, the purpose of this step is to use an Android meta-model to describe the structure of the Android model.

As shown in Fig. 4, reverse engineering was used [17] to obtain the Android meta-model related to SNFs for the generation of Android project code using code from Android APIs. The Android meta-model is described using UML class diagrams. In an Android meta-model, activities comprise a set of views, which could be UI widgets or layout components. In the case of UI widgets (*e.g.*, `Button`, `ListView`, `ImageView`, `TextView`), which extend `View` class can be used to register listeners in order to identify events triggered by users and determine whether navigation is involved. Intent is used to describe the source and destination associated with the triggered navigation event.

Table 2. Model-to-Model transformation rules.

Rule #	XMI tag in State Machine Model	Attribute	XMI tag in Android Model	Attribute
1	state	–	Activity	–
2	UI Component	Type	View	type
3	UI Component	VariableName	View	variableName
4	trigger	–	Listener	–
5	transition	–	Intent	–

As shown in Table 2, the definition of an Android meta-model results in a set of model-to-model transformation rules for the conversion of State Machine diagrams to Android models. In this study, ATLAS Transformation Language (ATL) [18] is used to implement the transformation rules. List 2 shows an example rule specified for the transformation of elements, including states, UI components, triggers, and transitions (defined in the State Machine meta-model) into corresponding elements in the Android meta-model.

List 2. Example of an ATL transformation rule

```
rule Element {
from
    s : StateMachine!State
    using{
        uiType : Sequence(String) = s.getUIType();
        uiVariable : Sequence(String) = s.getUIVariable();
        uiListener : Sequence(String) = s.getUIListener();
        uiTargetName : Sequence(String) = s.getTargetName();
    }
to
    ToActivity : AndroidModel!Activity(
        name <- s.name,
        View <- ToUi
    ),
    ToUi : distinct AndroidModel!UI_Component foreach(e in uiType)(
        type <- e,
        variableName <- uiVariable,
        Listener <- ToListener
    ),
    ToListener : distinct AndroidModel!Listener foreach(e in uiListener)(
        listenerName <- e,
        Intent <- ToIntent
    ),
    ToIntent : distinct AndroidModel!Intent foreach(e in uiTargetName)(
        targetName <- e,
        sourceName <- s.name
    )
}
```

Android models are generated in XMI format after the application of transformation rules. List 3 presents an example Android model derived from the State Machine model described in List 1, following the application of model-to-model transformation rules.

List 3. Example of Android model derived from List 1

```
<Activity name="Login">
    <View type="Button" variableName="loginBtn">
        <Listener listenerName="onClick">
            <Intent targetName="Main" sourceName="Login"/>
        </Listener>
    </View>
<Activity>
```

3.4 Design Model-to-Text Transformation Rules

The generated Android models are then transformed into source code specific to the Android platform. In this step, we develop a set of model-to-text transformation rules (Table 3) for the conversion of Android models into executable Android project code.

Table 3. Model-to-text transformation rules.

Rule #	XMI tag in State Android Model	Attribute	Android Code
1	Activity	–	Activity Name
2	View	Type	UI Type
3	View	VariableName	uiComponent
4	Listener	–	Listener
5	Intent	–	Intent, Source Activity, Target Activity

Unlike the previous step in which transformations between models are executed using a script-based rule language, in this step we leverage code templates to specify model-to-text transformations. Model-to-text transformation rules embedded in code templates are implemented using the Acceleo model transformation tool [19] in order to obtain executable Android project code from the Android models. List 4 presents an example of an Acceleo code template. List 5 presents an example of the Android Java code generated from List 2.

List 4. Example of Acceleo code template

```
    [comment encoding = UTF-8 /]
    [module generateActivity('http://org/model/AndroidModel')]
    [template public generateActivity(anActivity : Activity,aApplication : Application)]
    [file (anActivity.name.concat('.java'), false, 'UTF-8')]
    package [aApplication.packageName/];
    import android.app.Activity;
    import android.os. Bundle;
    import android.content.Intent;
    import android.widget.*;
    import android.view.View;
    public class [anActivity.name/] extends Activity {

    [for( view : View | anActivity.view)]
        [view.Type+' '+view.VariableName/];
    [/for]

    @Override
       protected void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.[anActivity.name.toLowerCase().concat('_activity')/]);

      [for(view : View| anActivity.view)]
        [view.VariableName/]=([view.Type/])findViewById(R.id.[view.VariableName/]);
        [view.VariableName/].set[view.Listener.ListenerName.
            toUpperFirst()/]Listener(new[view.Type/].
        [view.Listener.ListenerName.toUpperFirst()/]Listener(){

            public void [view.Listener.ListenerName/](View v) {
                Intent intent = new Intent();
                intent.setClass([view.Listener.Intent.sourceName/].this,
                [view.Listener.Intent.targetName/].class);
                startActivity(intent);
            }
        });
        [/for]
        }
    }
    [/file]
[/template]
```

List 5. Generated Android activity code

```
package com.example.myapplication;

import android.app.Activity;
import android.os.Bundle;
import android.content.Intent;
import android.widget.*;
import android.view.View;

public class Login extends Activity {

   Button loginBtn;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.Loginactivity);
        loginBtn =(Button)findViewById(R.id.loginBtn);
        loginBtn.setOnClickListener(new Button.OnClickListener(){
            public void onClick(View v) {
                Intent intent = new Intent();
                intent.setClass(Login.this, Main.class);
                startActivity(intent);
            }
        });
    }
}
```

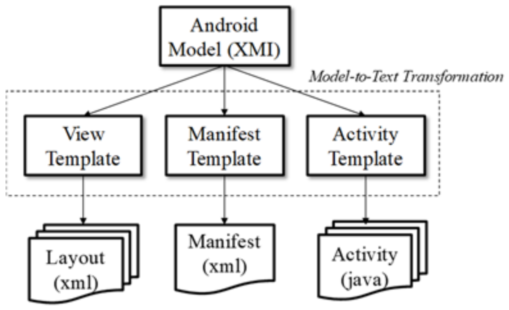


Fig. 5. Android project templates.

The generation of Android project code that can be imported to the Android IDE tool (*i.e.*, Android Studio) [20] requires other project-related artifacts in addition to Android Java code. As shown in Fig. 5, we developed templates for the required artifacts used to generate the corresponding project code. Application of these templates allows the transformation of the models of SNFs into Android project code and the required artifacts. The resulting Android project code can be deployed directly to Android smartphones, *i.e.*, without the need for manual intervention.

4. EXPERIMENTS

In this section, we describe the experiment results obtained using the proposed system. We first designed a synthetic experiment in which the proposed system was used to generate code from 25 State Machine models with various numbers of states and transitions. We then conducted a manual experiment in which 15 developers implemented an Android project involving 5 screens and 4 transitions. The time required to complete the two experiments was compared in order to assess the effectiveness of the proposed scheme. The experiments are detailed in the following:

4.1 Synthetic Experiment for Code Generation

In this experiment, we used multiple input UML models with various number of states and transitions. Each input model had two parameters: the number of layers (*LN*) and the number of transitions under each state (*STN*). Let us consider the input State Machine model as a tree, where *LN* denotes the depth of the tree, and *STN* denotes the breadth of all nodes except the leaf-nodes. An example model where $LN = 2$ and $STN = 2$ is presented in Fig. 6. The total number of states associated with the model is computed as follows:

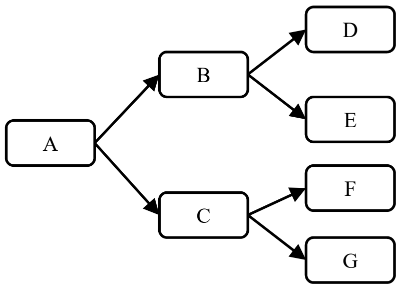


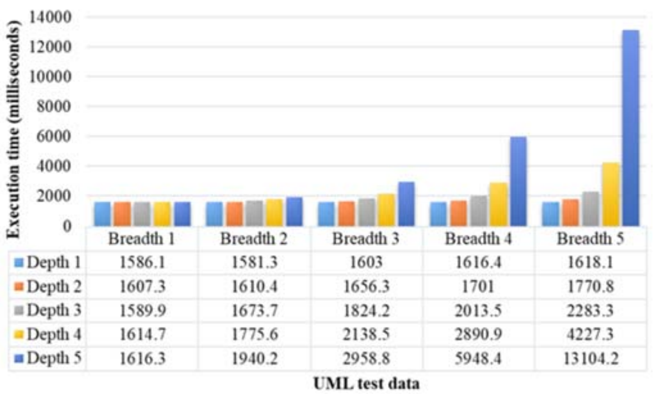
Fig. 6. Example state machine model where LN = 2 and STN = 2.

$$\textit{Total number of states} = 1 + STN^1 + \ldots + STN^{LN}$$

where 1 is the initial state and STN^i is the number of states in the ith layer.

For this experiment, we designed multiple State Machine models with depth and breadth varying from 1 to 5. This resulted in 25 test cases, in which the simplest case had only 2 two states and the most complex case included 3906 states. The experiment was executed on a standalone computer equipped by an Intel I5-3470 CPU, 16GB of main memory, and a 7200-rpm HDD (1TB).

The experimental results in Fig. 7 show that the execution time was less than 2 seconds in most test cases, and the most complex case required only 12 seconds. This is considerably faster than manual coding.



	Breadth 1	Breadth 2	Breadth 3	Breadth 4	Breadth 5
■ Depth 1	1586.1	1581.3	1603	1616.4	1618.1
■ Depth 2	1607.3	1610.4	1656.3	1701	1770.8
■ Depth 3	1589.9	1673.7	1824.2	2013.5	2283.3
■ Depth 4	1614.7	1775.6	2138.5	2890.9	4227.3
■ Depth 5	1616.3	1940.2	2958.8	5948.4	13104.2

Fig. 7. Time required to generate code for synthetic testing models.

4.2 Manual Coding Experiment

To measure the time required for the implementation of a simple project without the proposed scheme, we designed a simple Android project with 5 states and 4 transitions, as shown in Fig. 8. The time it took a team of 15 coders to program the project was carefully recorded (Fig. 9). The average time to complete the task was 1021 seconds (*standard deviation* = 299.8 *seconds*). The data in Figs. 7 and 9 indicates that the proposed system reduced coding time by an average of 15 minutes for a simple task within 5 states (*i.e.*, 3 minutes per state). A comparison of the time consumed in the two experiments revealed that the proposed MDA method can greatly reduce app development time through increased efficiency.

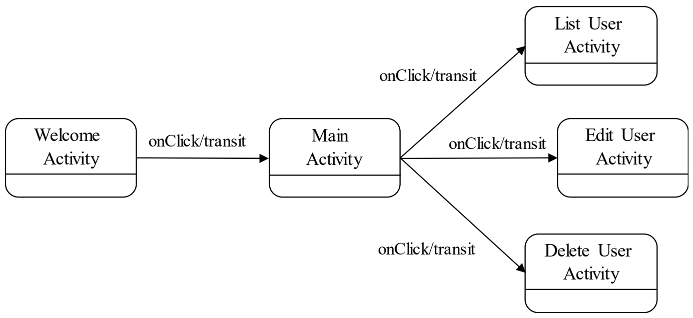


Fig. 8. Testing model for manual implementation.

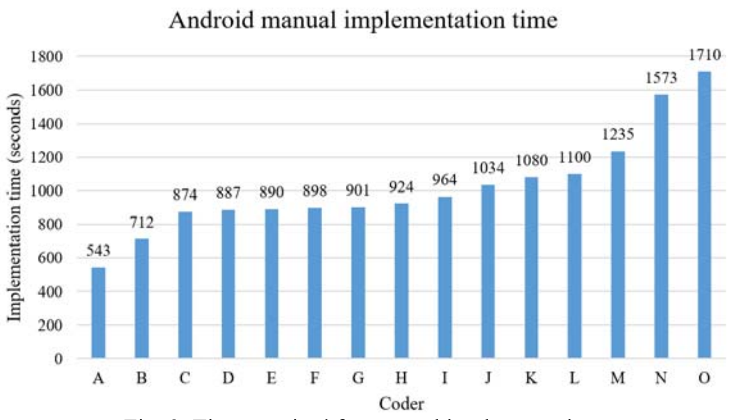


Fig. 9. Time required for manual implementation.

5. CONCLUSIONS

In this study, we developed an MDA-based app development scheme that takes SNFs as input for the automated generation of Android project code with all required artifacts. This method effectively reduces app development time. The proposed scheme ensures that the resulting Android project code conforms to the intended functionality, *i.e.* the model of SNFs. Implementation of the proposed scheme enables app developers to easily obtain app skeletons as well as all necessary UI components and screen navigability through the automated generation of code from wireframes. One limitation to the proposed scheme is the need for developers to implement the logic part of each screen, since this is not described in wireframes.

In future work, we intend to leverage the models of SNFs as test cases to verify whether the corresponding implementations adhere to the modeled navigation flows. Furthermore, we plan to tailor the proposed scheme to other app platforms, such as iOS.

REFERENCES

1. IDC Research, "Smartphone OS market share," http://www.idc.com/promo/smartphone-market-share/os, 2017.
2. N. Gandhewar and R. Sheikh, "Google Android: An emerging software platform for mobile devices," *International Journal on Computer Science and Engineering*, Vol. 1, 2010, pp. 12-17.
3. M. H. Goadrich and M. P. Rogers, "Smart smartphone development: IOS versus Android," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 2011, pp. 607-612.
4. N. Hertz, "Complete overview of the mobile app development process," *TECTINASIA*, 2015, txlabz.com/2015/12/.
5. C. Y. Wong, C. W. Khong, and K. Chu, "Interface design practice and education towards mobile apps development," *Procedia – Social and Behavioral Sciences*, Vol. 51, 2012, pp. 698-702.
6. R. Soley and the OMG Staff Strategy Group, "Model driven architecture," White paper 3.0, Object Management Group, 2000.
7. A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Longman Publishing, MA, 2003.
8. B. Unhelkar, *Software Engineering with UML*, Auerbach Publications, UK, 2017.
9. M. Ko, Y.-J. Seo, B.-K. Min, S. Kuk, and H. S. Kim, "Extending UML meta-model for Android application," in *Proceedings of the 11th International Conference on Computer and Information Science*, 2012, pp. 669-674.
10. A. G. Parada and L. B. de Brisolara, "A model driven approach for Android applications development," in *Proceedings of Brazilian Symposium on Computing System Engineering*, 2012, pp. 192-197.
11. S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for Android, automated software engineering," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 658-668.
12. M. Lachgar and A. Abdali, "Generating Android graphical user interfaces using an MDA approach," in *Proceedings of the 3rd IEEE International Colloquium in Information Science and Technology*, 2014, pp. 80-85.
13. A. G. Parada, E. Siegert, and L. B. de Brisolara, "Generating java code from UML class and sequence diagrams," in *Proceedings of Workshop de Sistemas Embarcados*, 2011, pp. 99-101.
14. H.-M. Chen, Y. T. Chen, Y. Ho, and Y. X. Yan, "AR-based taxi recommendation by leveraging crowd sharing comments," in *Proceedings of the 15th International Symposium on Pervasive Systems, Algorithms and Networks*, 2018, pp. 337-339.
15. D. Drusinsky, *Modeling and Verification Using UML Statecharts*, Elsevier, 2011.

16. Object Management Group, "The XML metadata interchange specification version 2.5.1," *Object Management Group*, 2015.
17. R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf, "A study on the current state of the art in tool-supported UML-based static reverse engineering," in *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002, pp. 22-32.
18. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: A QVT-like transformation language," in *Proceedings of the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems*, *Languages*, *and Applications*, 2006, pp. 719-720.
19. J. Musset, E. Juliot, and S. Lacrampe, *Acceleo User Guide*, Obeo Network, 2008.
20. K. Mew, *Mastering Android Studio 3: Build Dynamic and Robust Android Application*, Packt Publishing, UK, 2017

Hsi-Min Chen (陳錫民) received the B.S. and Ph.D. degrees in Computer Science and Information Engineering from National Central University, Taiwan, in 2000 and 2010, respectively. He is currently an Associate Professor with the Department of Information Engineering and Computer Science, Feng Chia University, Taiwan. His research interests include software engineering, object-oriented technology, service computing, and distributed computing.

Tsung-Chi Lin (林宗祺) received the B.S. degree in the Department of Information Engineering and Computer Science, Feng Chia University, Taiwan. He is currently a master student in the same department at Feng Chia University. His research interests include software engineering, mobile application technology, and education technology.

Lien-Wu Chen (陳烈武) received the Ph.D. degree in Computer Science and Information Engineering from the National Chiao Tung University, Hsinchu, Taiwan, in December 2008. From 2012 to 2015, he was an Assistant Professor with the Department of Information Engineering and Computer Science, Feng Chia University. Since August 2015, he is currently an Associate Professor. His research interests include wireless communication and mobile computing, especially in the Internet of Vehicles, Internet of Things, and Internet of People.

Bao-An Nguyen (阮寶恩) received the M.S. degree in Information Engineering and Computer Science from Feng Chia University, Taiwan, in 2011. He is currently pursuing the Ph.D. degree in the same department. His research interests include data mining, software engineering and education technology.

Yi-Chung Chen (陳奕中) received the B.S. and M.S. degrees in Electrical Engineering from National Cheng Kung University, Tainan, Taiwan, in 2007 and 2008, and the Ph.D. degree in Department of Computer Science and Information Engineering from National Cheng Kung University, Tainan, Taiwan, in 2014. He is currently an Assistant Professor in the Department of Industrial Engineering and Management, National Yunlin University of Science and Technology. His research interests include spatio-temporal databases, recommendation systems, social network analyses, artificial intelligences, and techniques of Industry 4.0.