

AMERICAN UNIVERSITY OF BEIRUT

A HIGH-LEVEL MODELING LANGUAGE FOR THE
EFFICIENT DESIGN, IMPLEMENTATION,
AND TESTING OF ANDROID APPLICATIONS

by
JOHN ABOU-JAOUDEH

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
September 28, 2015

AMERICAN UNIVERSITY OF BEIRUT

A HIGH-LEVEL MODELING LANGUAGE FOR THE
EFFICIENT DESIGN, IMPLEMENTATION,
AND TESTING OF ANDROID APPLICATIONS

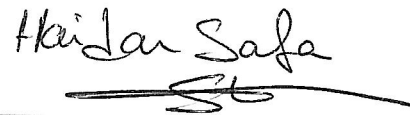
by
JOHN ABOU-JAOUDEH

Approved by:



Dr. Mohamad Jaber, Assistant Professor
Computer Science

Advisor



Dr. Haidar Safa, Associate Professor
Computer Science

Member of Committee



Dr. Marcel Karam, Associate Professor
Computer Science

Member of Committee

Date of thesis defense: August 20, 2015

AMERICAN UNIVERSITY OF BEIRUT

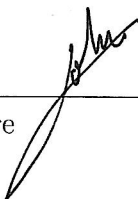
THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Abou Jaoudeh John Issam
Last First Middle

Master's Thesis Master's Project Doctoral Dissertation

I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project**, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

Signature 

Date 28/09/2015

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Mohamad Jaber who guided me all throughout the thesis. Thank you for sharing your knowledge, and experience with me.

And, I want to thank Dr. Yliès Falcone, Mr. Kinan Dak Al Bab, and Mr. Mostafa El-Katerji for their help.

I also want to acknowledge my committee members for their advice which was extremely helpful, and precise.

In addition, I must thank Dr. Wassim El Hajj, I appreciate your support all throughout my Computer Science masters degree at the American University of Beirut.

I would also like to thank Mr. Mustapha Hammam who was always there when I needed advice.

Last but not least, a thank you to my family for the support you've given me. I don't know how I could have weathered everything without you by my side. The love, and concern you've all showed truly means a lot to me. I am forever grateful.

AN ABSTRACT OF THE THESIS OF

John Abou-Jaoudeh for Master of Science
Major: Computer Science

Title: A High-Level Modeling Language for the Efficient Design, Implementation,
and Testing of Android Applications

Smartphones global penetration is on the rise, and currently covers more than quarter of the globe's population. Yet, developing mobile applications remains difficult, time consuming, and error-prone, in spite of the number of existing platforms and tools. In this report, we define MoDroid, a high-level modeling language to ease the development of Android applications. MoDroid allows the development of models which represent the core of applications. MoDroid provides Android programmers with the following advantages: (1) Models are built using high-level primitives that abstract away many implementation details allowing application development to be divided over several types of developers; (2) It allows the definition of interfaces between models to automatically compose them, which facilitates testing, and code reusability; (3) Java native android can be automatically generated along with the required permissions thus increasing performance, security, and privacy; (4) It supports efficient model-based testing that operates on models. MoDroid has been fully implemented and was used to develop several non-trivial Android applications. Moreover, MoDroid was compared against current market tools.

CONTENTS

ACKNOWLEDGEMENTS	v
ABSTRACT	vi
LIST OF FIGURES	ix
LIST OF TABLES	x
1 INTRODUCTION	1
1.1 Simple native Android application example	2
1.2 Issues faced when developing Android applications	5
1.2.1 Issues when designing an Android application	5
1.2.2 Issues when testing an Android application	6
1.3 Contributions.	6
1.4 Report organization.	7
2 RELATED WORK	9
2.1 Introduction	9
2.2 Modeling Frameworks	10
2.3 Mobile Development Frameworks	10
2.3.1 Native Development	10
2.3.2 Hybrid Development	11
2.3.3 Web-Based Development	11
2.4 Visual Development Tools	12
2.4.1 Comparison	13
2.5 Android Application Testing	15
2.5.1 GUI Based Testing	15
2.5.2 Non-GUI Based Testing	16
3 THE ANDROID META-MODEL	17
3.1 Introduction	17
3.2 LibModel	17
3.3 LibActivity	18
3.4 GUI Elements	18
3.4.1 LibView	18
3.4.2 Controls	19
3.4.3 Layouts	19
3.5 Handlers	21
3.6 Resource Management	23

4	PROJECTS COMPOSITION	24
4.1	Introduction	24
4.2	Principles	24
4.3	Example	26
4.4	Other Advantages	27
5	PERMISSION AUTO-DETECTION AND GENERATION	28
5.1	Introduction	28
5.2	Motivating Example	28
5.3	Proposed Solution	29
6	MODEL-BASED TESTING	32
6.1	Introduction	32
6.2	Model-Based Testing Module	33
6.2.1	LibTest	33
6.2.2	Text Manipulation	33
6.2.3	Click and Long Click Actions	33
6.2.4	Activity Manipulation	34
6.2.5	Activity Fields	34
6.2.6	Exception <code>ElementNotFoundException</code>	34
6.3	Example	35
7	CODE GENERATION	36
7.1	Definition and Usage	36
7.2	Implementation	36
7.3	Cloud-based Compilation	37
8	TOOL-SET - MODROID	39
8.1	Introduction	39
8.2	Implementation	39
8.3	Design-Flow Development	42
8.3.1	Development of Models	42
8.3.2	Composition of Models	42
8.3.3	Project Generation	42
9	EXPERIMENTAL RESULTS	43
9.1	Introduction	43
9.2	Code Length	43
9.3	Performance Testing Evaluation	44
9.3.1	Scientific Calculator Benchmarks	44
9.3.2	Volleyball Statistics Benchmarks	45
10	CONCLUSION AND FUTURE WORK	47
10.1	Conclusion	47
10.2	Future Work	48

LIST OF FIGURES

1.1	Screenshot from the simple application	2
2.1	Screenshot from Appery’s visual tool	13
3.1	Basic elements of the Meta-Model	21
4.1	Example of models composition.	26
7.1	Cloud-based Compilation	38
8.1	Development design-flow in MoDroid.	41
9.1	Volleyball Statistics Application Screenshot	46

LIST OF TABLES

2.1	Comparison of available tools.	14
9.1	Code length comparison.	44
9.2	Testing time scientific calculator (in seconds).	45
9.3	Testing time volleyball statistics (in seconds).	46

CHAPTER 1

INTRODUCTION

Contents

1.1	Simple native Android application example	2
1.2	Issues faced when developing Android applications	5
1.2.1	Issues when designing an Android application	5
1.2.2	Issues when testing an Android application	6
1.3	Contributions.	6
1.4	Report organization.	7

Smartphones have become a big part of our daily life. Whether to communicate, read articles, listen to music, watch videos, track our fitness, or any other task we do, it involves our smartphone. Smartphones are categorized into brands such as: Samsung, Apple, Nokia, Motorola, Sony, Blackberry. Each of these brands comes with a default platform/operating system installed. Android is the most popular platform for mobile devices, with over 84% of the market share at the end of 2014.

Developing native applications on Android platform requires a developer to be an expert in Java and the Android SDK; the Android software development kit (SDK) contains libraries to build Android applications, and without this SDK one cannot create an application. There are several toolsets available to guide the development of applications; toolsets such as Eclipse and Android Studio. Both contain graphical tools which can help the developer with the GUI side of an application, and a source editor where Android code can be written. Section 1.1 below shows an example of a simple android application.

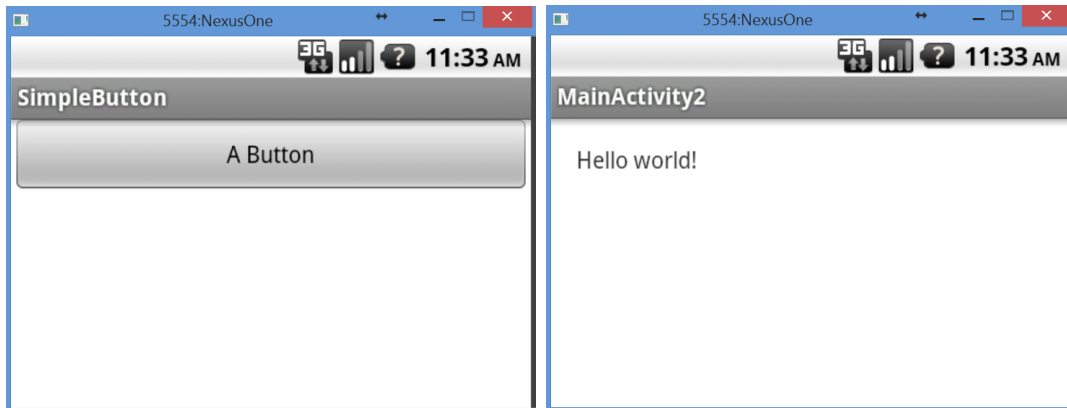


Figure 1.1: Screenshot from the simple application

1.1 Simple native Android application example

This simple application contains two activities; an activity in Android represents a window/frame in the GUI. The first activity contains a button, which, when clicked on, navigates us to the second activity. The second activity contains a text box which displays `Hello world!` to the user. Figure 1.1 shows a screenshot from this application.

Below is the code for the first activity `MainActivity` whose layout is a `LinearLayout`, and contains a button with text `A Button`, which, when clicked on, starts the second activity `MainActivity2`.

```

1 package com.example.simplebutton;
2 import android.os.Bundle;
3 import android.app.Activity;
4 import android.content.Intent;
5 import android.view.Menu;
6 import android.view.View;
7 import android.view.ViewGroup.LayoutParams;
8 import android.widget.Button;
9 import android.widget.LinearLayout;
10 public class MainActivity extends Activity {
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         LinearLayout parent = new LinearLayout(this);
15         parent.setLayoutParams(new LinearLayout.LayoutParams(LayoutParams.MATCH_PARENT, LayoutParams.WRAP_CONTENT));
16         parent.setOrientation(LinearLayout.HORIZONTAL);
17         Button bt = new Button(this);
18         bt.setText("A Button");
19         bt.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.WRAP_CONTENT));
20         bt.setOnClickListener(new View.OnClickListener() {
21             public void onClick(View v) {
22                 Intent myIntent = new Intent(MainActivity.this, MainActivity2.class);

```

```

23     MainActivity.this.startActivity(myIntent);
24     }
25     });
26     parent.addView(bt);
27     setContentView(parent);
28 }
29
30 @Override
31 public boolean onCreateOptionsMenu(Menu menu) {
32     getMenuInflater().inflate(R.menu.main, menu);
33     return true;
34 }
35 }

```

Next, we need to develop the second activity. We notice that in the first activity we generated the layout programatically. In the second activity, we will generate the layout using an XML file which can be built using the visual tools in Eclipse and Android Studio.

The second activity, MainActivity2, in this case is contains the minimal code required to create an activity. We notice that we set the content view to be the XML file activity_main_activity2.

```

1 package com.example.simplebutton;
2 import android.os.Bundle;
3 import android.app.Activity;
4 import android.view.Menu;
5 public class MainActivity2 extends Activity {
6     @Override
7     protected void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.activity_main_activity2);
10    }
11    @Override
12    public boolean onCreateOptionsMenu(Menu menu) {
13        // Inflate the menu; this adds items to the action bar if it is present.
14        getMenuInflater().inflate(R.menu.main_activity2, menu);
15        return true;
16    }
17 }

```

Now we generate the XML file which represents the GUI of this activity.

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"

```

```

        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context=".MainActivity2" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/hello_world" />
    </RelativeLayout>

```

Finally, we have the `AndroidManifest.xml` file which is the configuration file of the applications. It contains the list of activities and other Android components available in this application. Any access to a device feature (such as GPS, camera, etc.) requires a permission which must be manually added to this file.

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.simplebutton"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="18" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity android:name="com.example.simplebutton.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="com.example.simplebutton.MainActivity2"
            android:label="@string/title_activity_main_activity2" >
        </activity>
    </application>
</manifest>

```

1.2 Issues faced when developing Android applications

Various frameworks/tools have been developed to facilitate the development of applications, each of which has its own pros and cons. Yet, creating a correct and efficient Android application remains a difficult endeavor for several reasons that can be categorized under design or testing issues.

1.2.1 *Issues when designing an Android application*

First, the programming model in Android involves different components (e.g., `Activity`, `Service`, `BroadcastReceiver`, `ContentProvider`, etc.), with a complex interaction model between these components (e.g., `Handler`, `Intent`, etc.). Second, to separate the internal representation of information from its presentation to the user, most of the frameworks supporting the development process use the Model-View-Controller (MVC) design pattern to split an application into three interconnected parts. However, as applications become more complex, the MVC pattern must be augmented with a new paradigm that guides developers on how to split the core of an application into different interconnected parts. Such paradigm shall facilitate and encourage the concurrent development of an application by several developers. Third, Android provides a protection mechanism to device-specific features (e.g., GPS, camera, vibrator, internet, SMS, address book, SD card, etc.) by offering a specific set of programmatic APIs to access them. Then, the application configuration file (`AndroidManifest.xml`) must explicitly include access permissions for all features that are used within the application. At installation, the application is given permission to the corresponding features (from the configuration file) and the user will be aware about the required permissions. If an application calls an API to access a specific feature that requires a permission access and the configuration file does not contain that access permission, a runtime exception will be raised at the start-up of the application. Clearly, users prefer applications with minimum set of permissions. This protection mechanism is often error-prone and in most of the

cases developers end up using permissions they do not require in their code, or the opposite [Bartel et al.(2012)Bartel, Klein, Le Traon, and Monperrus].

1.2.2 Issues when testing an Android application

Ensuring that applications are performing as required has become more challenging given the daily dynamic change in the domain of mobile technology. Application users mainly face problems of the following kind: incorrect behavior, crashes, and Application becoming Not Responsive (ANR), etc. Keeping in mind the complexity of mobile application development, and the inability to eliminate bugs and errors, an essential component of mobile development is testing. The process of Mobile Application Testing is used to detect the errors that might have occurred during the development of the application, to ensure that user expectations are met, and to make sure that applications have been executed properly. This is essential to be done by application developers who aim to keep their customers satisfied, and entertained by the final product.

1.3 Contributions.

The challenges of programming mobile applications have prompted us to reconsider the best practices of their design development. For this purpose, a framework with the following features is desirable: (1) the framework should abstract away different implementation details; (2) decompose the development process into different stages; and (3) include automated code manipulation and generation. To do so, we define a Meta-Model for the development of mobile android applications. Meta-modeling drastically improves flexibility of development, hence allows us to manage applications more easily.

The Meta-Model consists of a set of modules that represent Graphical User Interfaces (GUIs) and their respective handlers in an abstract and a simpler way than Native Java Android. We implement the Meta-Model along with several modules

in MoDroid to tackle the aforementioned problems. MoDroid contains the following modules:

1. A composition module takes as input Android Java models and the connections between them. The composition module allows to easily parallelize the development process.
2. A permission analysis that automatically discovers the required permissions of an application.
3. A code generator which automatically generates native Android Java code given an android Java model.
4. An activity-builder module automatically builds an activity in the Android Java model given an XML file representing that activity.
5. An efficient model-based testing that allows to easily write test cases using high-level primitives and to efficiently execute them.

Our framework facilitates and speeds-up the development process. It transforms an Android application into an Android Java model that is compliant to the Meta-Model and contains all the necessary information about the application. The current version of our Meta-Model covers a subset of Android API that includes all the main constructs and functionalities. Consequently, it is designed with backward compatibility in mind so that developers can write native Android code within the model to use features currently not covered within the Meta-Model.

1.4 Report organization.

The rest of this report is structured as follows. Chapter 4 presents the Meta-Model. The following sections present the components associated to the Meta-Model: model composition is presented in Chapter 5; and automatic permissions detection is presented in Chapter 6; model-based testing framework is presented in Chapter 7; and

automatic code generation (from high-level model to native android) is presented in Chapter 8. Chapters 9 and 10 describe MoDroid, a full implementation of our framework and some benchmarks. Chapter 3 discusses related work. Chapter 11 draws some conclusions and perspectives.

CHAPTER 2

RELATED WORK

Contents

2.1	Introduction	9
2.2	Modeling Frameworks	10
2.3	Mobile Development Frameworks	10
2.3.1	Native Development	10
2.3.2	Hybrid Development	11
2.3.3	Web-Based Development	11
2.4	Visual Development Tools	12
2.4.1	Comparison	13
2.5	Android Application Testing	15
2.5.1	GUI Based Testing	15
2.5.2	Non-GUI Based Testing	16

2.1 Introduction

This report advocates the use of modeling to improve the development of Android applications. In this section we first discuss some of the available modeling frameworks, we then go into existing Android development frameworks, and finally we put forward the current market testing tools.

2.2 Modeling Frameworks

Modeling parts of an application simplifies and accelerates the development process and frees the developer from writing repetitive code.

The use of models in the development of Java applications has received a lot of attention, and several tools are available. For instance, Eclipse Modeling Framework (EMF) [Steinberg et al.(2003)Steinberg, Budinsky, Merks, and Paternostro] is a powerful modeling tool based on two metamodels Ecore, and Genmodel. EMF stores the model information using XMI (XML Metadata Interchange), and creates its meta-model via UML, Java annotations, XML Schema, and XMI. Similarly, Xcore [Foundation(2011)], another tool from Eclipse, is a textual syntax for Ecore.

Both EMF and Xcore are powerful tools when it comes to modeling Java applications. However, to the best of our knowledge they have not been used to develop Android applications.

2.3 Mobile Development Frameworks

Mobile development frameworks are usually categorized into native, cross-platform, and web based. In addition to the above categories, there are visual tools which provide a graphical interface that facilitates the development.

A native mobile development framework creates applications in native code. Each of those categories has its advantages, and disadvantages. For example, native has the best performance, while visual tools and web based allow for the fastest development. We compare our approach with some of the frameworks in those categories.

2.3.1 *Native Development*

Native applications naturally, do not use HTML and CSS; development is done by writing code in the native language of the platform, and using the libraries

provided by this platform. In our case, for a developer to produce a native Android application, the code must be written in Java, and using Android SDK.

Applications developed using the native language use a device's resources more efficiently, and thus excel in performance compared to hybrid, or web based applications.

On the other hand, hybrid and web-based frameworks are not developed using the native language. The code is compiled into a mobile application. This allows for the same code to be compiled into several platforms (cross platform development); therefore drastically reduces the development time of applications as compared to writing the native language of each platform alone to generate the same application but on different operating systems.

2.3.2 Hybrid Development

PhoneGap [Systems(2009)] and Cordova [Apache(2011)] are two commonly used cross platform mobile development frameworks [Palmieri et al.(2012)Palmieri, Singh, and Cicchetti]. They allow the developer to generate mobile applications that work on almost all devices by using HTML, CSS, and JavaScript. Using JavaScript to interact with the phone's features prevents from using native code since JavaScript is slower in processing data. Moreover, these frameworks lack the ability for background processing, which might be important in several applications. This could be improved with the use of Web Workers which is currently not compatible with a wide range of browsers. Furthermore, performance issues were reported due to the lack of hardware CSS acceleration of Android [Wolf and HUFFSTADT(2013)].

2.3.3 Web-Based Development

jQuery mobile [jQuery Team(2010)] is one of the most used web based mobile development frameworks. It allows for extremely rapid development of responsive web

sites, and applications which can be accessed via all smartphone, tablet, and desktop devices.

Another trending framework which is commonly used is Bootstrap [Otto and Thornton(2011)]. Bootstrap is an HTML, CSS, and JavaScript framework for developing responsive, mobile first web applications.

Two main disadvantages arise when using web based frameworks: poor performance [Rösler et al.(2014)Rösler, Nitze, and Schmietendorf], and losing the ability to use smartphone features.

2.4 Visual Development Tools

App Inventor 2 [Mitchell(2014)] is a GUI-based tool which supports the rapid development for simple applications. However, when it comes to complex applications, App Inventor 2 sets a lot of limits on the developer, and on the application itself since users cannot write their own code, and are only limited to what is provided by the GUI.

On the other hand, MobiA [Balagtas-Fernandez and Hussmann(2008)] propose a model driven graphical modeling language for mobile application development. It is easy to use, but does not support model composition, nor does it have a dedicated testing model.

Furthermore, Appery [Appery(2010)] provides a cloud-based mobile app builder which is GUI based. It allows the creation of pages and services. In addition to the GUI based tool, Appery allows the developer to write their own Javascript and CSS files manually if needed, and to use all the features present in Adobe Cordova [Apache(2011)]; giving the developer more control over the application. The project may be exported as Android, iOS, and Windows Phone applications; the project may also be exported as an HTML/JS/CSS website. Figure 2.1 shows a screenshot of the interface used to develop applications.

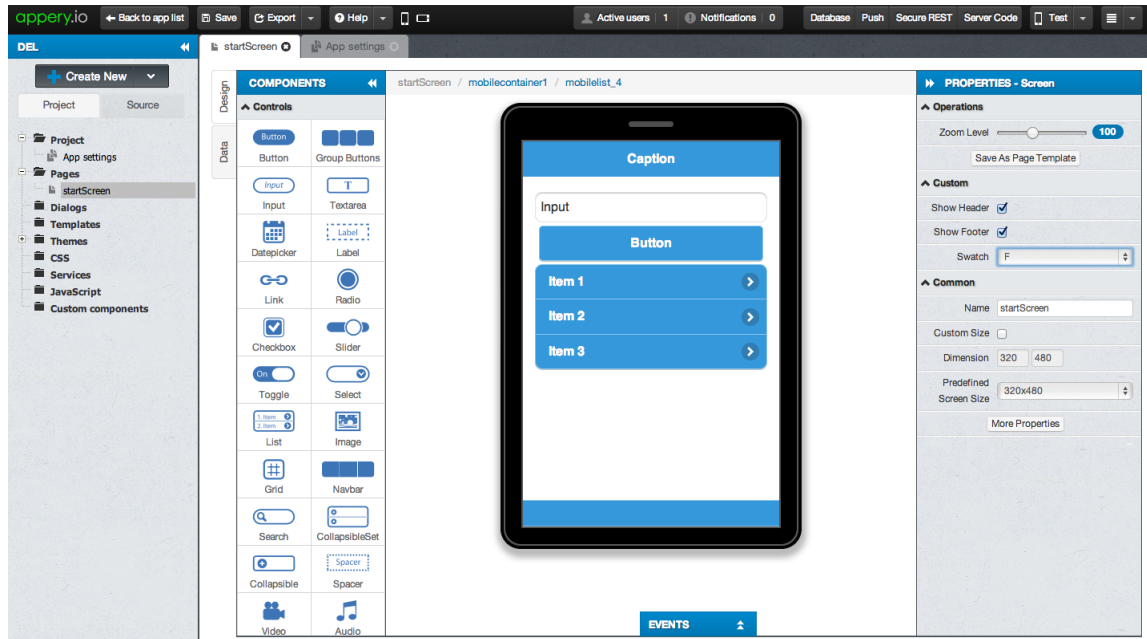


Figure 2.1: Screenshot from Appery’s visual tool

Appery is not the only cloud-based tool; we could find tens of similar tools. These tools share the disadvantages of hybrid and web-based frameworks when it comes to performance, and the use of phone features.

2.4.1 Comparison

None of the above Android development frameworks allows for the composition and decomposition of applications. Our framework allows for this, as shown in Section 5. Moreover, it allows for permission auto-detection and generation as specified in Section 6. The main advantage is that any unneeded permission will not be included in the Android Manifest file allowing the application to be available for more devices, and most importantly protecting the user’s privacy when using additional unneeded permissions [Feng et al.(2014)Feng, Anand, Dillig, and Aiken] [Au et al.(2012)Au, Zhou, Huang, and Lie].

Table 2.1 shows a comparison between the frameworks and tools discussed above.

	MoDroid	Native	PhoneGap	Cordova	jQuery Mobile	Bootstrap	Appery
Language	Java	Java	HTML/ CSS/JS	HTML/ CSS/JS	HTML/ CSS/JS	HTML/ CSS/JS	HTML/ CSS/JS
Ease of Development	✓	X	✓	✓	✓	✓	✓
Cross-Platform	X	X	✓	✓	✓	✓	✓
Access to Features	✓	✓	Limited	Limited	Limited	Limited	Limited
Ease of Decomposition	✓	X	X	X	X	X	X
Permission Auto-Detection	✓	X	Adds Unneeded	Adds Unneeded	Uses Browser	Uses Browser	Adds Unneeded
Hardware Acceleration	✓	✓	X	X	X	X	X
Background Processing	Future Work	✓	✓	Compatibility Issues	Compatibility Issues	Compatibility Issues	Compatibility Issues
Testing Tool	✓	✓	✓	✓	✓	✓	✓
Model Based Testing	✓	X	X	X	X	X	X
Testing Efficiency	✓	X	✓	✓	✓	✓	✓
Testing Simplicity	✓	X	X	X	X	X	X

Table 2.1: Comparison of available tools.

2.5 Android Application Testing

Testing of android applications has become more challenging. In general, android testing tools can be divided into two main categories: GUI based testing and non-GUI based testing.

2.5.1 GUI Based Testing

This category requires testing on an emulator or on a real android device. Google presents several tools some of which fall under this category. First is Instrumentation [Google(2007)], a set of classes and methods which control Android components and how Android loads applications. These classes allow the developer to test any component at any given time in its lifecycle. Developing a test case with this tool is time consuming and very complex. This lead Google to develop another tool Espresso [Google(2013)]. Espresso is built over Instrumentation and its main goal is to simplify testing techniques.

Another commonly used tool is Robotium [Reda(2009)]. This tool is well documented and could be easily configured. In addition to the above, developing test cases is simple; all action calls are being done on a single object `solo`. The main disadvantage one would face using this tool is the speed of running test cases.

Other tools under this category parse applications and automatically generate test cases, e.g., Monkey [Google(2010)], Android GUI-TAR [Amalfitano et al.(2014)Amalfitano, Fasolino, Tramontana, Ta, and Memon] and ORBIT [Yang et al.(2013)Yang, Prasad, and Xie].

Whether on an emulator or on a real android device, running an enormous number of test cases would require a huge amount of time (see Section 10). This would make GUI based testing tools fall a lot behind non-GUI based testing tools. On the other hand, GUI based testing is more expressive and would be useful to test hardware devices (e.g., camera, sensors, etc.).

2.5.2 Non-GUI Based Testing

Robolectric [Labs(2010)] allows developers to test Android applications without the use of an Android emulator or device. Robolectric presents the user with several objects and methods to imitate an android application's lifecycle. The main advantage is the speed of running test cases. We would be able to perform thousands of operations by the time GUI based testing is able to perform just tens.

Configuring this tool as well as writing test cases are complicated and time consuming. Moreover, it is dependent on several other libraries. For instance, Listing 2.1 is a sample code to access the value of a `TextView` using Robolectric.

Listing 2.1: Sample code to access the value of a `TextView` using Robolectric.

```
1 ActivityClassName activity =  
2     Robolectric.buildActivity(ActivityClassName.class).create().start().visible().get();  
3 TextView results = (TextView) activity2.findViewById( viewID );  
4 results.getText();
```

Our framework falls under the category of non-GUI based testing. We target ease of configuration, simplicity and performance.

CHAPTER 3

THE ANDROID META-MODEL

Contents

3.1	Introduction	17
3.2	LibModel	17
3.3	LibActivity	18
3.4	GUI Elements	18
3.4.1	LibView	18
3.4.2	Controls	19
3.4.3	Layouts	19
3.5	Handlers	21
3.6	Resource Management	23

3.1 Introduction

The Meta-Model consists of a set of modules used to model the core of an Android application. The Meta-Model allows to model an Android application as a Java object. The modeling process abstracts away implementation details. Moreover, the resulting object model can be easily and efficiently manipulated by applying model transformation and composition as described in the remainder of this report.

3.2 LibModel

The Meta-Model consists of a hierarchy of classes. The top element of the hierarchy is the project: `LibModel`. Each instance of this type represents an independent application. A `LibModel` consists of a set of activities mapped to names, global variables, and meta-information related to the project. Listing 3.1 shows how a `LibModel` is initialized.

Listing 3.1: Example of a LibModel inside the BMI module.

```
1 LibModel bmiModel = new LibModel("bmiModel", "health.app", "John");
```

3.3 LibActivity

An activity `LibActivity` is the android equivalent of a window or frame. The developer can create instances of `LibActivity`, fill it up with GUI elements, and then add it to a `LibModel`. A `LibActivity` can contain GUI elements (e.g., layout, button, etc.), packaging information, and *activity scope variables*. The developer can also provide methods for handling events related to the activity's life cycle: `onCreate`, `onStop`, etc. Moreover, `LibActivity` has a constructor that takes an XML file as argument containing a view description of the activity and automatically instantiates the corresponding object. That is, we can still benefit from MVC design pattern supported for native android development. Listing 3.2 gives an example on how to create a `LibActivity` object, and how to add this activity inside a `LibModel`.

Listing 3.2: Example of a LibActivity inside the BMI module.

```
1 LibActivity userInputActivity = new LibActivity();
2 //Initialize the LibActivity Object
3 ...
4 //Implement the activity's layout and actions
5 bmiModel.addActivity(userInputActivity, "userInputActivity");
6 //Add the activity to the module
7 ...
```

3.4 GUI Elements

3.4.1 LibView

GUI elements, also called views, are the building blocks of an application. All GUI elements inherit their basic attributes from `LibView`, an abstract class that contains the basic attributes and methods for the manipulation of appearance of an

element such as width, height, padding, etc. Views are categorized into Controls, and Layouts. A view can be either added to an Activity or to a layout.

3.4.2 Controls

The controls currently provided by the Meta-Model, prefixed with `Lib`, are the following: `Button`, `ImageButton`, `TextView` (equivalent to a `Label`), `TextField`, `ToggleButton` (on/off button), `Spinner` (similar to drop-down list), `RadioButton`, `CheckBox`, etc. Listing 3.3 presents how a `LibTextView` object is initialized, and also shows how to change the `TextView`'s width, and text.

Listing 3.3: Example of a `LibTextView` inside the BMI module.

```
1 LibTextView heightLabel = new LibTextView();
2 heightLabel.setWidth(LibView.MATCH_PARENT);
3 heightLabel.setText("Height (cm)");
```

3.4.3 Layouts

Layouts are special views that can contain other views. They control the position of the view within the activity. A layout is treated as a `View`. It has its own attributes such as width, height, and others. It can be added to activities, or to other layouts. The layouts provided by the Meta-Model, prefixed with `Lib`, are the following: `LinearLayout` (views are placed in order in a line; can be horizontal, or vertical), `RadioGroup` (a `LinearLayout` that acts as a `RadioButton` group as well), `FrameLayout` (displays all views in the same position above each other), `RelativeLayout` (controls the position of views by using them as anchors), and `TableLayout` (organizes the views into rows and columns). Listing 3.4 gives an example of a `LibLinearLayout` object creation with orientation settings. It also shows how to add a `View(LibTextView)` to this layout, and finally how to set this layout as an activity's main view.

Listing 3.4: `LibLinearLayout` BMI Module Example.

```

1 LibLinearLayout layout = new LibLinearLayout();
2 layout.setOrientation(LibLinearLayout.ORIENTATION.vertical);
3 ...
4 //Adding a view to a layout
5 layout.addView( heightLabel );
6 ...
7 //Setting an activity's layout
8 userInputActivity.setView( layout );
9 ...

```

These views cover all the basic elements of Android applications. Moreover, it is possible to extend the Meta-Model by adding more views in an easy and modular way. Figure 3.1 depicts the basic elements of the Meta-Model.

Hereafter, we show a step-by-step example on how to build a simple health application using our paradigm. The health application consists of two basic modules: (1) Body Mass Index (BMI); and (2) Menu Planner/Meal Planner. The BMI module is composed of two activities. The first activity manages user inputs (weight and height) and computes the BMI. Then, it sends the computed value to the second activity. If the user does not enter a value and clicks on compute, the phone vibrates signaling an error. Moreover, the user inputs are stored in the activity scope variables. The second activity is where the BMI value is displayed. From this activity, a user may either navigate back to activity one or navigate to Menu Planner/Meal Planner module. Listing 3.5 shows a snapshot of the code of BMI module.

Listing 3.5: Snapshot of the code of BMI module.

```

1 LibModel bmiModel = new LibModel("bmiModel", "health.app", "John");
2 LibActivity userInputActivity = new LibActivity();
3 LibActivity resultActivity = new LibActivity();
4 bmiModel.addActivity(userInputActivity, "userInputActivity");
5 bmiModel.addActivity(resultActivity, "resultActivity");
6 setUserInputActivityLayout( userInputActivityLayout );
7 setResultActivityLayout( resultActivityLayout );
8 ...

```

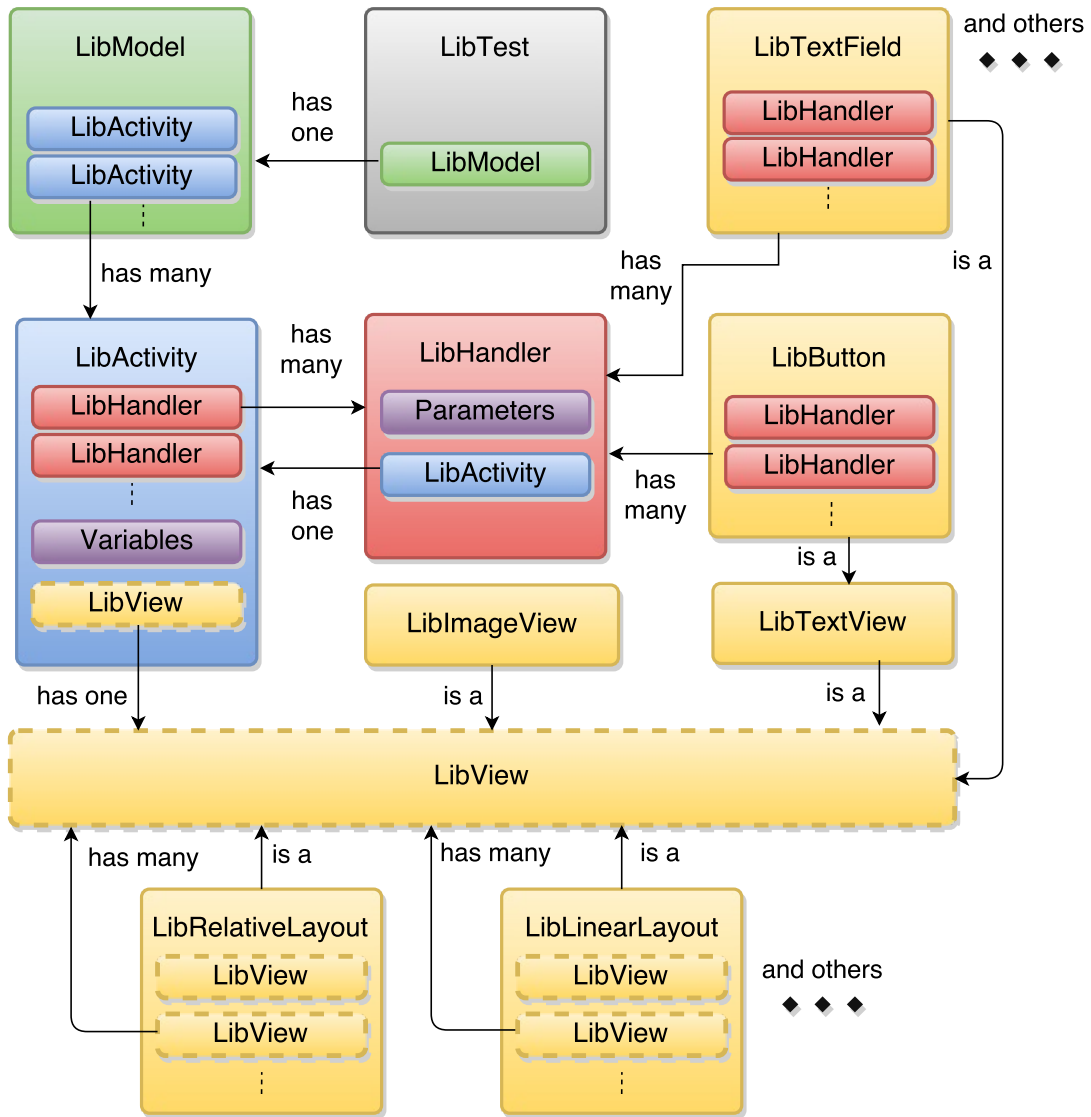


Figure 3.1: Basic elements of the Meta-Model

3.5 Handlers

Some views have special events that trigger specific handlers (e.g., on button click). A developer can either write a method which handles the event or use some pre-defined shortcuts. The code within the handlers can either use functionalities of the Meta-Model or directly use native Android code. Views can be accessed within handlers by passing them as parameters of the handler method. A handler can be used for the communication between activities. For example, when a button is clicked or some text field gets modified, one common functionality is to go to another activity.

For a given view, one specifies its handler method by calling `setOnClickListener`. The Meta-Model simplifies control transfer by using high-level shortcut. For instance, within a handler, `startActivity` method redirects to another activity by taking the name of the activity and any view objects as parameters. Another shortcut is to directly specify the next activity in the `setOnClickListener`.

Data parameters can be sent with a control transfer to communicate between activities. These parameters can be passed either as parameters (1) to `startActivity` along with the next activity; or (2) directly to `setOnClickListener`.

Listing 3.6 shows the code of the button from the first activity where its handler computes the BMI value and sends it to the second activity. Note that, if the user does not enter a value and clicks on compute, the phone vibrates signaling an error.

Listing 3.6: Example of a handler with data transfer.

```
1 calculateButton.setOnClickListener("Handler:health.BMI.calculate", height, weight);
2
3 // package health.BMI
4 public void calculate(LibView ht, LibView wt) {
5     if(!ht.getText().equals("") && !wt.getText().equals("")) {
6         double val = computeBMI (ht, wt);
7         LibModel.startActivity("resultActivity", val);
8     }
9     else {
10        Vibrator v = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
11        if(v.hasVibrator()) v.vibrate(500);
12    }
13 }
```

These parameters can be accessed in the main method by using a special formatted string (`@param_{i}` to get the i^{th} parameter). Within a handler, these parameters can be also accessed by calling `LibActivity.getParameter(i)` to get the i^{th} parameter. Listing 3.7 shows a snapshot of the code that sets some of the views of the second activity. It sets the value of a text view to the passed parameter

that comes from the first activity. Also, it uses a shortcut to set the handler of the button that redirects to the first activity.

Listing 3.7: Example of shortcut handler and data access.

```
1 bmiValueText.setText("@param_0");
2 ...
3 goBackButton.setOnClickListener("GoToActivity:userInputActivity");
```

3.6 Resource Management

One of the most effort consuming tasks in developing Android applications is resource management: images, application icons, and other types of resources. These resources are copied to specific folders within the resource folder. In our Meta-Model, resources are automatically added and generated into their corresponding folders. For example, to use an image, the developer only needs to add the path of the image/icon to be used. Listing 3.8 shows an example that specifies the icon of an application, displays an image, and creates a button with an image displayed.

Listing 3.8: Example of resource management.

```
1 ...
2 model.setIcon("images/application.jpg"); // sets the application icon
3 // Create a label to display the given image.
4 LibImageView imageView = new LibImageView("images/image.jpg", ...);
5
6 // Create a button with an image displayed on it.
7 LibImageButton imageButton = new LibImageButton("images/button.jpg", ...);
```


CHAPTER 4

PROJECTS COMPOSITION

Contents

4.1	Introduction	24
4.2	Principles	24
4.3	Example	26
4.4	Other Advantages	27

4.1 Introduction

Decomposing projects into smaller parts is a key concept in software engineering. Using the Meta-Model, it is possible to develop several models and automatically compose them according to a user-provided configuration. Additionally, developers may create models, and use them across all their applications; for example, a login model. The composition operation takes as input a configuration file that specifies the links between the interfaces of models. Each link specifies some control and data transfer that have to take place upon the occurrence of an event in the models: the activity from another project that has to be executed and the parameters that have to be sent.

4.2 Principles

Given n models m_1, m_2, \dots, m_n , where m_i consists of $a_1^i, a_2^i, \dots, a_{I_i}^i$ activities. Recall that each activity has views that may have handlers. Each handler runs some code that may transfer the control to another activity that can be an identified activity in the model or a symbolic activity (i.e., an activity which is identified by a symbolic value). Symbolic activities within a handler are specified by using method

`goToUnknown` that takes an identifier and a set of objects (to be passed to the other activity) as parameters. A model that has a handler that transfers to a symbolic activity is considered as a *partial model*.

If a handler only redirects to a symbolic activity, it is possible to use pre-defined high-level shortcut to do so. At an abstract level, the composition module relies on two functions: interface that returns the symbolic activities in a model, and, configuration that associates (concrete) activities to symbolic activities. The definition of function interface is obtained by an automatic analysis of models (see Chapter 9). Function configuration is defined by the user through a configuration file. A configuration file is of the form depicted in Listing 4.1.

Listing 4.1: General shape of a configuration file.

```

1 <New Project Name>
2 <New Project Package>
3 <New Project Author>
4 <Model>.<Activity>; //indicates the main activity of the composed project
5 <Model>.<Unknown ID> -> <Model>.<Activity Name>; // mapping
6 <Model>.<Unknown ID> -> <Model>.<Activity Name>; // mapping
7 <Model>.<Unknown ID> -> <Model>.<Activity Name>; // mapping
8 ...

```

It first contains the new project name, package, author and main activity. Then, it defines the mapping between identifiers and activities of different models. Let m_i be a partial model with some of its handlers associated to symbolic activities id_1^i, id_2^i ($\text{interface}(m_i) = \{id_1^i, id_2^i\}$).

For example, Let a_k^j be an activity of model m_j , one can have $\text{configuration}(id_1^i) = a_k^j$, which means that identifier id_1^i of model m_i is mapped to activity a_k^j of model m_j .

Figure 4.1 is an example of two partial models M_1 and M_2 . The handler of button `button2`, a handler of activity A_2 and the handler of button `button3` redirect to symbolic activities through interfaces I_1 , I_2 and I_3 , respectively. The configuration file connects I_1 , I_2 and I_3 to activities A_5 , A_4 and A_1 , respectively.

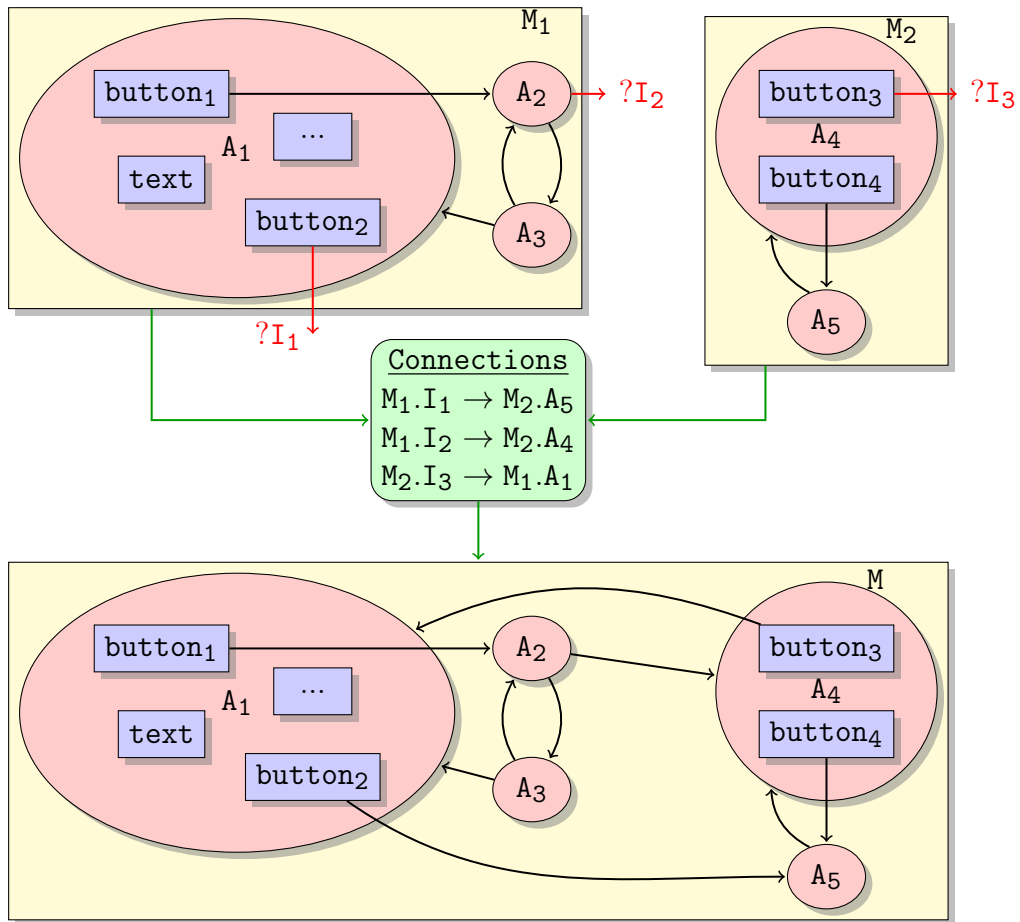


Figure 4.1: Example of models composition.

4.3 Example

Listing 4.2 shows a snapshot of the shortcut handler of the button from the second activity (result activity) of the health application that redirects to a symbolic activity of a different model through the interface `menuPlannerInterface`.

Listing 4.2: Example of unknown shortcut handler.

```
1 menuPlannerButton.setOnClickListener("GoToActivity:Unknowns(menuPlannerInterface)");
```

Finally, models can be composed to build the final project by using one of `LibModel`'s overloaded constructors that takes a configuration file and a set of models. The composition of BMI and Menu Planner modules is depicted in Listing 4.3.

Listing 4.3: Composition of BMI and Menu Planner modules.

```
1 LibModel healthAppModel = new LibModel("config.txt", bmiCalculatorModel, menuPlannerModel);
```

Listing 4.4 shows the configuration file that connects (1) the menu planner interface of BMI calculator model to user information activity of the menu planner model; and (2) the BMI calculator interface of menu planner model to user input activity of BMI calculator model.

Listing 4.4: Configuration file connecting BMI and Menu Planner models.

```
1 Health App // project name
2 health.app // project package
3 John // project author
4 bmiCalculatorModel.userInputActivity // main activity of the composed model
5 // connections/mapping
6 bmiCalculatorModel.menuPlannerInterface -> menuPlannerModel.userInformationActivity
7 menuPlannerModel.bmiCalculatorInterface -> bmiCalculatorModel.userInputActivity
```

Note that, a set of models can be composed successively to build the final model. Listing 4.5 shows an example of successively composing three models.

Listing 4.5: Successive composition of models.

```
1 LibModel model12 = new LibModel("config1.txt", model11, model12);
2 LibModel model123 = new LibModel("config2.txt", model12, model13);
```

4.4 Other Advantages

Although mobile applications almost certainly harbor undetected errors, using models composition approach, it is possible to directly apply software testing paradigm to reduce and locate them: unit and integration testing. This can be done by testing partial models separately (unit testing) to find local errors and then test the complete model (integration testing) to find interface errors.

CHAPTER 5

PERMISSION AUTO-DETECTION AND GENERATION

Contents

5.1	Introduction	28
5.2	Motivating Example	28
5.3	Proposed Solution	29

5.1 Introduction

Manually managing permissions in the configuration file is time consuming. It often entails several compilation attempts of the application to narrow down the proper set of required permissions. Consequently, most of the developers add permissions more than it is needed which contradicts the users' preferences.

5.2 Motivating Example

To use the phone's vibrator, one needs to retrieve the vibrator object using the method `getSystemService(Context.VIBRATOR_SERVICE)`, then call one of the following methods: `hasVibrator()`, `vibrate()`, or `cancel()`. Note that, method `hasVibrator()` returns a boolean and does not require the `vibrate` permission (`android.permission.vibrate`), while `cancel()` and `vibrate()` do. Listing5.1 shows an example of native Android Java code that calls `hasVibrator()` but does not require permission access which is actually not needed. Intuitively, developers may assume that method `hasVibrator()`, or/and class method `getSystemService()` requires permission `android.permission.VIBRATE` and adds it to the manifest configuration file. Note that, if one replaces line 8 with `v.vibrate(500)`, the permission access would be required only for mobiles that

have a vibrator. Consequently, code modifications demand a manual reconsideration of the required permissions.

Listing 5.1: Example of native Android Java code that does not require permission.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_main);
5
6     Vibrator v = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
7     if(v.hasVibrator()) {
8         Toast.makeText(context, message, duration).show();
9     }
10 }
```

5.3 Proposed Solution

In our case, APIs which access device-specific features are called within handlers of listener GUI elements. Note that some external libraries may call some of these APIs. Our permission detection/generation module must take into account: (1) modification (add/remove/update) of permissions; (2) modification (add/remove/update) of APIs; (3) modification (add/remove/update) of external library that may call those APIs.

In other words, any of these modifications should not drastically affect the code that automatically detects and generates permissions. For this, we define a set of templates that represent all the APIs that require a permission. For instance, object initializations (constructors), method calls (method name, parameter types, calling object's type), etc.

This gives us maintainability for future permission modification as well as ease to extend our supported set of permissions. We define two types of templates `permissions.xml` and `permissionExternals.xml` that contain templates for native APIs and external library APIs, respectively, that require permission access. The template file is of the form depicted in Listing 5.2.

Listing 5.2: General shape of a template file for a given permission.

```
<permission name="PERMISSION_1">
  <class name="Class_1">
    <method name="method_1">
      <parameters>
        <parameter type="param1" />
      </parameters>
    </method>

    <method name="method_2">
      <parameters>
        <parameter type=" " />
      </parameters>
    </method>

    <method name="method_3">
      <parameters>
        <parameter type="param2" />
        <parameter type="param3" />
      </parameters>
      <parameters>
        <parameter type="param4" />
      </parameters>
    </method>
  </class>

  <class name="Class_2">
    <method name="method_4" />
    <method name="method_5" />
  </class>

  <class name="Class_3" />
</permission>
```

The template depicted in Listing 5.2 defines all the API calls shown in 5.3 that require permission PERMISSION_1:

Listing 5.3: API calls requiring permission PERMISSION_1.

```
2 (Class_1).method_1(param1);
```

```

4 (Class_1).method_2();

6 (Class_1).method_3(param2, param3);

8 (Class_1).method_3(param4);

10 (Class_2).method_4(/*Any set of parameters*/);

12 (Class_2).method_5(/*Any set of parameters*/);

14 Class_3 var = new Class_3(/*Any set of parameters*/);

```

For example, the template for permission `android.permissions.VIBRATE` is depicted in Listing 5.4.

Listing 5.4: Template for permission `android.permissions.VIBRATE`.

```

<permission name="android.permissions.VIBRATE">
  <class name="Vibrator">
    <method name="vibrate" />
    <method name="cancel">
      <parameters>
        <parameter type=" " />
      </parameters>
    </method>
  </class>
</permission>

```

From the template of permission `android.permissions.VIBRATE`, we can deduce that permission `android.permissions.VIBRATE` is required whenever one of the lines of code in Listing 5.5 is detected.

Listing 5.5: API calls requiring permission `android.permissions.VIBRATE`.

```

1 // v is an object of type Vibrator
2 // E.g., Vibrator v = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
3
4 v.vibrate(500); // vibrate(long milliseconds) method
5
6 v.vibrate({{12}, {23}, {12}}, 50); // vibrate(long[] pattern, int repeat) method
7
8 v.cancel(); // cancel() method

```


CHAPTER 6

MODEL-BASED TESTING

Contents

6.1	Introduction	32
6.2	Model-Based Testing Module	33
6.2.1	LibTest	33
6.2.2	Text Manipulation	33
6.2.3	Click and Long Click Actions	33
6.2.4	Activity Manipulation	34
6.2.5	Activity Fields	34
6.2.6	Exception <code>ElementNotFoundException</code>	34
6.3	Example	35

6.1 Introduction

Performing test cases on an Android emulator or device is time consuming. Running hundreds of operations usually takes hours, if not days. Android SKD does not allow for test cases to be executed directly from inside an IDE, thus throwing a `RuntimeException` error.

In order to integrate efficient model-based testing in our framework, we extend our model to be executable. That is, each model can be represented as a state consisting of the current activity, the value of the views, the value of the activity scope variables and global variables. We implement all the functionalities to perform operations on any given MoDroid model. For example: (1) modify or get the value of a view; (2) perform click/event.

6.2 Model-Based Testing Module

6.2.1 *LibTest*

The model-based testing framework consists of a module `LibTest` that allows to perform high-level operations on the model under test (e.g., `setText`, `click`, etc.). `LibTest` takes a model under testing as input with an optional entry point (i.e., name of an activity) and a set of test cases to be performed.

Listing 6.1: LibTest Example.

```
1 ...  
2 LibTest test = new LibTest(bmiModel);  
3 ...
```

Recall that, it is possible to test partial models separately (unit testing) to find local errors and then test the composed model (integration testing) to find interface errors. Below we present some of the operations that can be performed on an Android model.

6.2.2 *Text Manipulation*

The developer can use `LibTest` to get, and modify a view's text using the `getText`, and `setText` methods respectively. An `ElementNotFoundException` is thrown when the view being used is not found.

Listing 6.2: Text Manipulation Example.

```
1 ....  
2 test.setText("weight", "70");  
3 assertEquals("Incorrect Weight", "70", test.getText("weight"));  
4 ...
```

6.2.3 *Click and Long Click Actions*

`LibTest` allows the developer to test click, and long click actions on a view using two methods: `click`, and `longClick`. An `ElementNotFoundException` is thrown if

the view's `click/longClick` functionality wants to change the current activity into an activity which does not belong to the model.

Listing 6.3: Click Example.

```
1 ...
2 test.click("calculateButton");
3 ...
```

6.2.4 Activity Manipulation

`LibTest` contains methods to manage activities; these methods allow a tester to use functionalities such as to get current activity name, and set the current activity. These kind of actions might throw an `ElementNotFoundException` when the activity specified does not belong to the model.

Listing 6.4: Activity Manipulation.

```
1 ...
2 assertEquals("Incorrect Activity", "resultActivity", test.getCurrentActivityName());
3 ...
```

6.2.5 Activity Fields

`LibTest` provides the method `getCurrentActivityFields` which returns the extra fields for the current activity.

6.2.6 Exception `ElementNotFoundException`

`ElementNotFoundException` is an exception used in `LibText` to signal that the developer is trying to access an element which is not available. The element can be a view, or an activity.

6.3 Example

Listing 6.5 shows an example of some test cases of BMI calculator model. It mainly tests the redirection of activities and the computation of BMI. It consists of the following steps:

1. Create a `LibTest` instance that takes the model as input. Note that, it is possible to specify an activity to be an entry point of the model.
2. Set the weight and the height values and check if the values have been set properly.
3. Perform click on `calculateButton` button and check if (1) the next activity is the result activity; and (2) the BMI was correctly computed.

Listing 6.5: Example of test cases.

```
1 @Test
2 public void testcase1(LibModel bmiModel) {
3     try {
4         LibTest test = new LibTest(bmiModel);
5         test.setText("height", "175");
6         assertEquals("Incorrect Height", "175", test.getText("height"));
7
8         test.setText("weight", "70");
9         assertEquals("Incorrect Weight", "70", test.getText("weight"));
10
11        test.click("calculateButton");
12
13        assertEquals("Incorrect Activity", "resultActivity", test.getCurrentActivityName());
14        assertEquals("Incorrect Value", "22.9", test.getText("value"));
15    } catch (ElementNotFoundException e) {
16        fail("Element Not Found: " + e);
17    }
18 }
```

CHAPTER 7

CODE GENERATION

Contents

7.1	Definition and Usage	36
7.2	Implementation	36
7.3	Cloud-based Compilation	37

7.1 Definition and Usage

Given an Android model we implement a module that generates equivalent native Android code (along with its resources, manifest configuration file, etc.). This is done by calling `generate(path)` method on a given model. The generated code preserves the order of statements and comments. This allows to easily integrate other functionalities to the generated code.

An example of code generation is depicted in Listing 7.1.

Listing 7.1: Example of code generation.

```
1 public class Application {
2     public static void main(String[] args) throws FileNotFoundException, IOException {
3         ...
4         healthAppModel.generate("gen/");
5     }
6 }
```

7.2 Implementation

The generate method recursively traverses the model shown in Figure 3.1 starting from `LibModel`, `LibActivity` instances, and all the way down to the layouts, and controls. Each object has its own generate method, which produces its equivalent native Android code.

Additionally, further methods written by an application developer are also parsed using ANTLR [Parr(2007)] and `StringTemplate` [Parr(2000)]. These additional methods are then translated into the equivalent native Android code. It is also possible to write native code inside these methods, where our permission detection module validates it against the defined templates, and gradually builds the `AndroidManifest.xml` configuration file with only the required permissions.

Finally, when model composition has been used to combine two or more models, the generation process correspondingly parses the configuration file which binds these two models. The result is one native application.

7.3 Cloud-based Compilation

Android SDK (Software Development Kit) is a set of components that include libraries, a debugger, a handset emulator, and others. Its main role in development is to generate native Android application executable files (.apk). Android SDK is a heavy module that requires memory, and time.

For this, we have developed a web service, and placed it online to generate an application's executable without installing the SDK. We have configured a server on the cloud with: (1) all the updated Android SDK libraries; (2) ant-apache which is a command based tool to create, and update an application given its source code; (3) compiled version of MoDroid. The web service takes as parameter a model of an Android application developed using MoDroid. The server compiles an application and generates an executable file (.apk) ready to be installed on Android devices, and shared on Google Play Store. As a plus, in order to efficiently test an application on different real devices, the web service, can send the generated executable to a list of email addresses (application beta testers).

Figure 7.1 illustrates the process of uploading a MoDroid Android Java code, and receiving an executable native Android code compiled by our server.

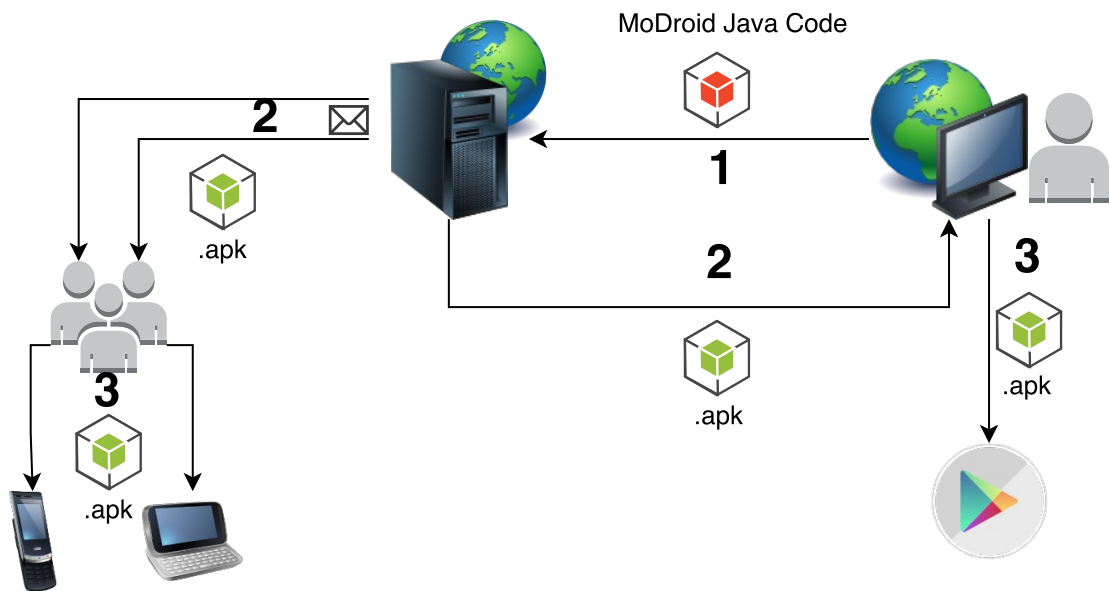


Figure 7.1: Cloud-based Compilation

CHAPTER 8

TOOL-SET - MODROID

Contents

8.1	Introduction	39
8.2	Implementation	39
8.3	Design-Flow Development	42
8.3.1	Development of Models	42
8.3.2	Composition of Models	42
8.3.3	Project Generation	42

8.1 Introduction

MoDroid¹ implements the Meta-Model and its supported tools: composition of models, permission detection, testing and code generation. The tool is packed and compiled into a single `jar` file. The `jar` file must be imported as a library to the project being developed.

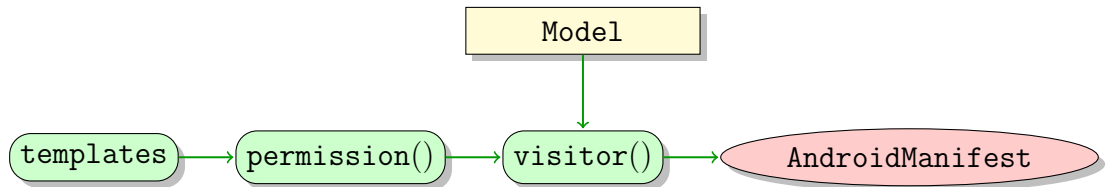
8.2 Implementation

To promote extensibility and modularity of MoDroid, we implement a visitor pattern that traverses the tree structure (GUI element, handlers, etc.) of an Android model. The pattern takes as input an interface that declares methods to be executed depending on the node that was localized. We have developed several implementations of that interface:

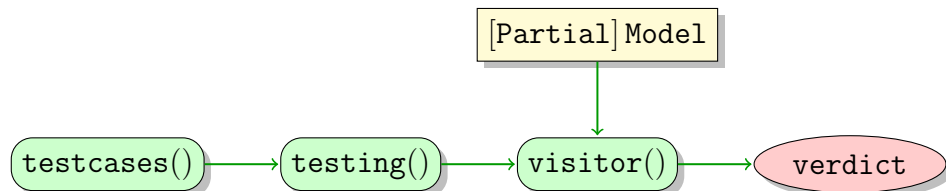
1. Implementation to detect unknown interfaces (symbolic activities) used in models composition.

1. <http://ujf-aub.bitbucket.org/modroid/>

- Implementation that takes templates representing all the APIs that require permissions and detect the required permissions accordingly.



- Implementation to make the model executable by performing operations on a view (e.g., `LibText`) that are used by model-based testing module. It recursively navigates into the layouts to reach the intended view, and then performs the action specified. Two executable actions are available. The first is `Clickable` action, where the reached view is clickable, and the action is to perform a click, or a long click. The second is `HasText` action, where the reached view has text, and the action is either to get the text, or to set the text of a view.



- Implementation to generate equivalent native Android code (along with its resources, manifest configuration file, etc.) from an Android model. Code generation module uses `antlr` and template engine library `StringTemplate` [Parr(2000)] for parsing handlers and generating native Java Android from an Android model.

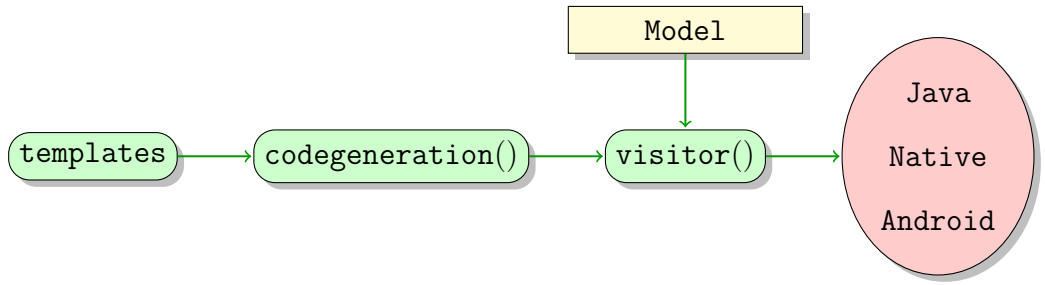


Figure 8.1 shows the development design-flow which is based on MoDroid.

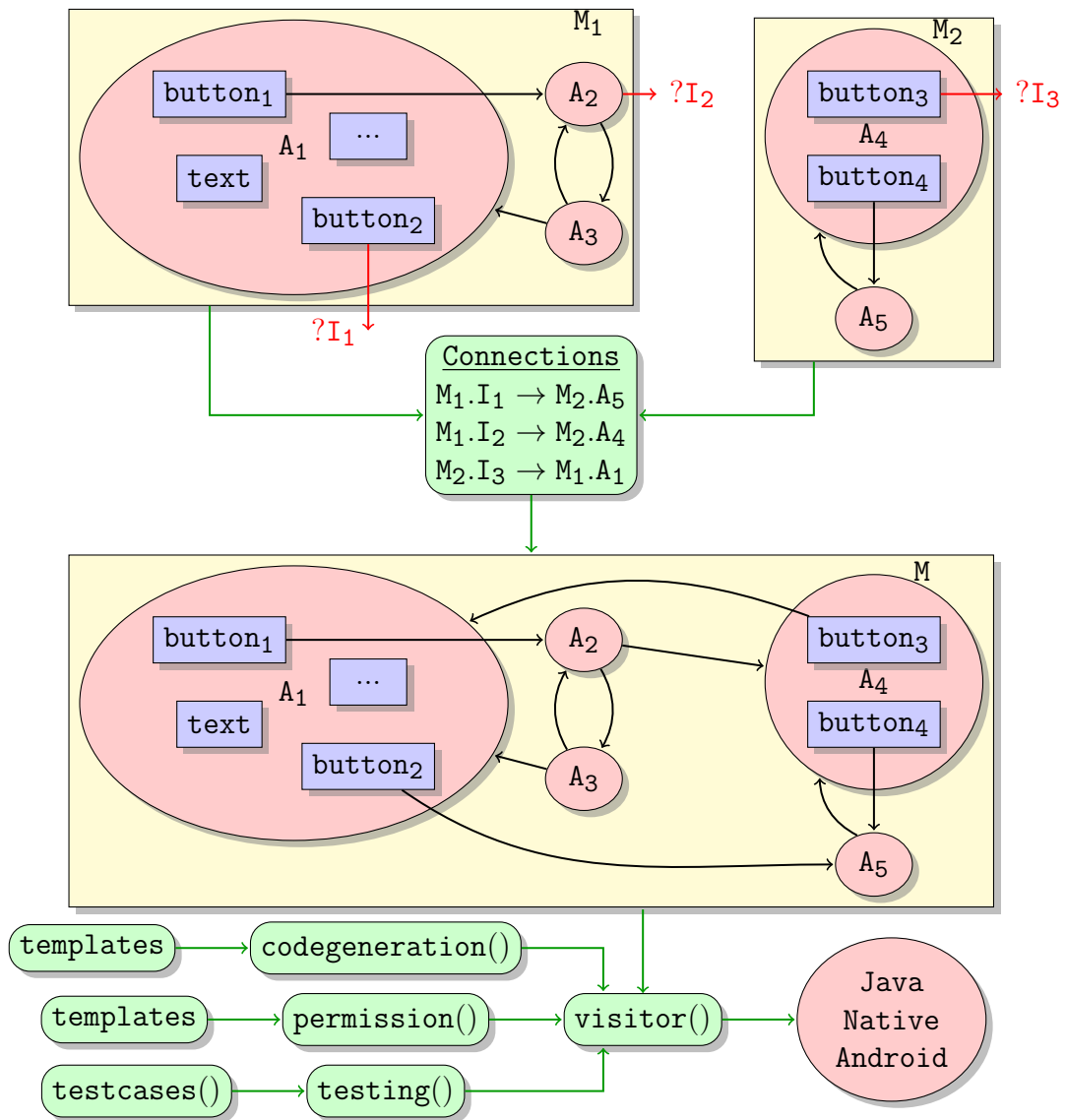


Figure 8.1: Development design-flow in MoDroid.

8.3 Design-Flow Development

8.3.1 *Development of Models*

Models are built and tested separately using high-level primitives provided by MoDroid. Recall that, it is also possible to build models without their handlers (e.g., only GUI layouts) from an XML file and then handlers can be programmatically integrated. That is, using the Meta-Model, one can still benefit from MVC design pattern supported for native Android development.

8.3.2 *Composition of Models*

Given a configuration file describing the mapping between models, we generate and test the final model of the application. It is worth mentioning that, we can build several applications given different mappings without any modifications of the models.

8.3.3 *Project Generation*

After executing the generate method, a native Java Android code is generated including all permissions that are required by the application core. Developers may edit the generated code by writing native Android code to add any extra functionalities before producing the Android Package Kit (.apk) file.

CHAPTER 9

EXPERIMENTAL RESULTS

Contents

9.1	Introduction	43
9.2	Code Length	43
9.3	Performance Testing Evaluation	44
9.3.1	Scientific Calculator Benchmarks	44
9.3.2	Volleyball Statistics Benchmarks	45

9.1 Introduction

We have developed several applications using both native Java Android and MoDroid; both versions of the code have the same design, and perform exactly the same functions. We then compared the code length of both. Furthermore, we developed test cases which used the current available tools, and we compared their performance with respect to our model-based testing tool. The results are displayed below.

9.2 Code Length

Table 9.1 compares the number of lines of code of several applications (Breadcrumb Viewer, Guessing Game, Scientific Calculator, and Volleyball Statistics) between native Java android, MoDroid, and automatically generated code. It is clear that building an Android model drastically reduces the number of lines of code. Moreover, it is much less time consuming with respect to writing native Java Android. We notice an overhead of 25% in the automatically generated code. This overhead is mainly due to the code generation of handlers. In fact we duplicate handlers of different views which can be technically eliminated by creating only one method for the same handler code of different views.

Application Name	MoDroid	Generated	Native	Written Code Reduction	Overhead Code Generated
Breadcrumb Viewer	63	329	276	77%	17%
Guessing Game	158	340	246	35%	27%
Scientific Calculator	180	377	282	36%	25%
Volleyball Statistics	137	702	510	73%	27%

Table 9.1: Code length comparison.

9.3 Performance Testing Evaluation

Moreover, we have conducted other benchmarks to compare the performance of our model-based testing framework and the following tools that are currently being widely used: Robolectric, Robotium, and Espresso on both an Emulator and a real device.

Robotium and Espresso perform actions on an emulator or on a real device; whereas Robolectric and MoDroid testing do not need an emulator nor a real device. Taking this factor into consideration, we would expect our testing framework and Robolectric to have a better performance.

9.3.1 *Scientific Calculator Benchmarks*

The first benchmark was performed on the scientific calculator application that we developed using MoDroid. The test actions were simply to click on values and operations; then to check the output of the calculator.

Table 9.2 shows a comparison of the time taken to perform test cases that require 10, 25, 50 up to 1 million operations by all the tools. Operations consist of performing clicks and text value modifications and searches.

As expected, Robolectric and MoDroid drastically outperform Robotium and Espresso. The results were close between Robolectric and MoDroid if we take into account the initialization phase required by Robolectric. The time taken to

Operations (#) Platform	10	25	50	75	100	150	1000	10000	100000	1000000
Robotium	36	88	180	268	360	541	3605.4	> 10 hours	> 10 hours	> 10 hours
Espresso Emulator	1.8	4.3	8.1	12	15.6	23.1	159.3	1645.5	16411.7	> 10 hours
Espresso Sony Z2	0.9	2.4	4.5	6.8	8.9	13.4	88.6	918.9	9189	> 10 hours
Robolectric	4.4	4.5	4.7	4.9	5.1	5.4	5.6	5.9	6.9	27
MoDroid	0.021	0.031	0.038	0.039	0.04	0.05	0.14	0.4	1.118	7.7

Table 9.2: Testing time scientific calculator (in seconds).

perform test cases requiring one million operations with Robolectric is 27 seconds as opposed to 7.7 seconds using MoDroid.

9.3.2 Volleyball Statistics Benchmarks

The second benchmark was performed on a Volleyball Statistics application developed using MoDroid. It is composed of two activities. The first activity is the splash screen which contains a button to navigate to the second activity where statistics are done. The second Activity is composed of two teams and the players for each team. Each player has two buttons to increment and decrement the points scored by this player. This application can be used by coaches, statistics frameworks, and so on. Figure 9.1 displays screenshots of both activities, one on a Sony Z2 device, the other on a Nexus One emulator.

We test this application by randomly selecting a player and performing operations. We also test the navigation between activities.

Table 9.3 shows a comparison of the time taken to perform test cases requiring 10, 25, 50 up to 1 million operations by all the tools. Similar to the first benchmark, Robolectric and MoDroid outperform other tools. Moreover, the time taken by test cases that require one million operations with Robolectric is 118 seconds as opposed to 12 seconds using MoDroid.

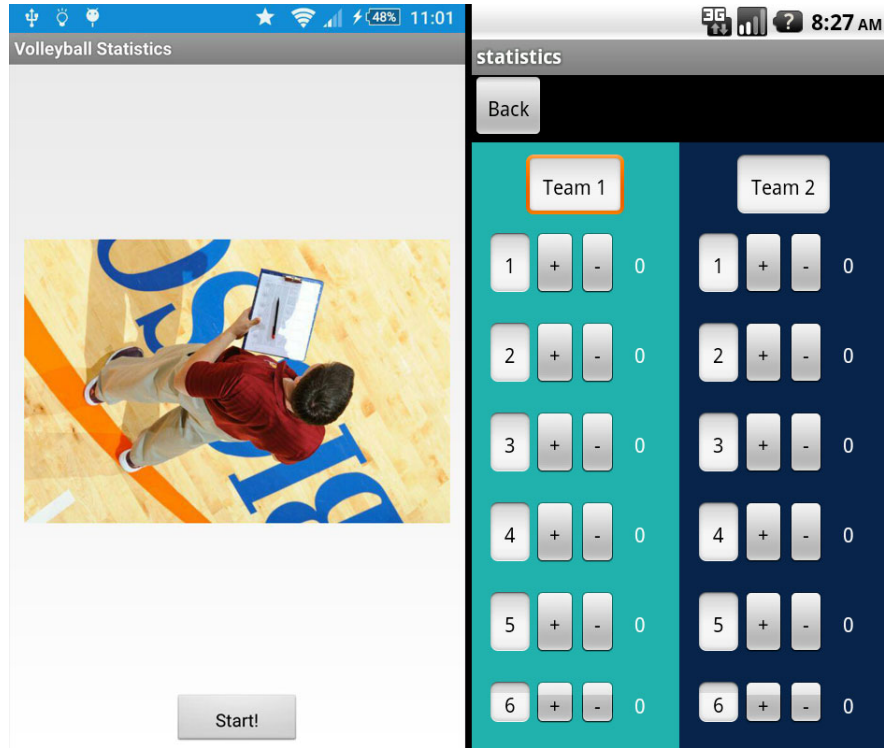


Figure 9.1: Volleyball Statistics Application Screenshot

Operations (#) \ Platform	10	25	50	75	100	150	1000	10000	100000	1000000
Robotium	5.2	12.8	25.8	37.1	49.9	73.5	486.1	4861.1	> 10 hours	> 10 hours
Espresso Emulator	3.1	7.6	15.5	22.5	29.9	43.6	290.9	2845.1	28760.2	> 10 hours
Espresso Sony Z2	1.1	2.6	5.5	7.7	10.3	15.3	111.9	1148.5	11275.2	> 10 hours
Robolectric	4.81	4.94	5.1	5.3	5.64	5.95	6.3	8.88	18.94	118.84
MoDroid	0.01	0.02	0.03	0.04	0.06	0.07	0.19	0.62	1.78	12.14

Table 9.3: Testing time volleyball statistics (in seconds).

CHAPTER 10

CONCLUSION AND FUTURE WORK

Contents

10.1 Conclusion	47
10.2 Future Work	48

10.1 Conclusion

This report proposes a new way to develop Android applications. It offers a compromise between expressiveness and ease of development: at the price of slightly reducing expressiveness, MoDroid facilitates and speeds up the development process. Yet, using our framework does not prevent developers from building applications using the full range of features of Android because, after automatically generating the base of the application, expert developers can still use Android features by completing the generated code template. Moreover, our framework introduces several interesting features for developers:

1. Decomposition of applications for parallel development and modularity; thus boosting code re-usability, and facilitating the option of having multiple developers on a certain application; as a result making development exponentially faster.
2. Automatic detection of permissions and generation of the `AndroidManifest` file. Consequently, making the application safer, and increasing users' privacy.
3. Efficient model-based testing of applications. Thus allowing a huge number of test cases to be executed in a reasonable amount of time.

4. Automatic code generation of some parts of applications which allows splitting application's development on several developers each having their own language skills. Moreover, allowing for native Android code to be added by expert Android developers.

10.2 Future Work

In the near future, we plan to add several features in the road-map of MoDroid.

1. Initially, we plan to add emulators for hardware components such as the GPS and camera. For instance, this should allow the user to pre-define GPS locations to be passed to the application.
2. Moreover, we plan to extend MoDroid to support a high-level description of multi-tasking, services, broadcast receivers, etc.
3. Additionally, we plan to make automatic permission detection compatible with the permissions model of the latest version of Android.
4. Furthermore, we plan to allow programmers to develop applications right on our cloud-based service via a web-based development kit without having to download any tool or library.
5. Finally, we plan to make MoDroid compatible with existing tools for automatic test generation for Android.

REFERENCES

- [Amalfitano et al.(2014)Amalfitano, Fasolino, Tramontana, Ta, and Memon] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. Mobiguitar – a tool for automated model-based testing of mobile apps. IEEE Software, NN(N):NN–NN, 2014.
- [Apache(2011)] Apache. Cordova, <http://cordova.apache.org/>, 2011. URL <http://cordova.apache.org/>.
- [Appery(2010)] LLC Appery. Appery.io, 2010. URL <http://www.appery.io>.
- [Au et al.(2012)Au, Zhou, Huang, and Lie] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 217–228. ACM, 2012.
- [Balagtas-Fernandez and Hussmann(2008)] Florence T Balagtas-Fernandez and Heinrich Hussmann. Model-driven development of mobile applications. In Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on, pages 509–512. IEEE, 2008.
- [Bartel et al.(2012)Bartel, Klein, Le Traon, and Monperrus] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 274–277. ACM, 2012.
- [Feng et al.(2014)Feng, Anand, Dillig, and Aiken] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In SIGSOFT FSE, 2014.

- [Foundation(2011)] Eclipse Foundation. Xcore is an extended concrete syntax for ecore that, in combination with xbase, transforms it into a fully fledged programming language with high quality tools reminiscent of the java development tools., 2011. URL <http://wiki.eclipse.org/Xcore>.
- [Google(2007)] Google. Testing instrumentation, 2007. URL <https://developer.android.com/tools/testing/index.html>.
- [Google(2010)] Google. Application exerciser monkey, 2010. URL <http://developer.android.com/tools/help/monkey.html>.
- [Google(2013)] Google. Espresso, 2013. URL <https://code.google.com/p/android-test-kit/wiki/Espresso>.
- [jQuery Team(2010)] jQuery Team. Jquery mobile, <http://www.jquerymobile.com/>, 2010. URL <http://www.jquerymobile.com/>.
- [Labs(2010)] Pivotal Labs. Robolectric, 2010. URL <http://www.robolectric.org/>.
- [Mitchell(2014)] Edward Mitchell. App Inventor 2: Tutorial: The fast and easy way to create Android apps, volume 1. Edward Mitchell, 2014.
- [Otto and Thornton(2011)] Mark Otto and Jacob Thornton. Twitter’s bootstrap, 2011. URL <http://getbootstrap.com/>.
- [Palmieri et al.(2012)Palmieri, Singh, and Cicchetti] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. Comparison of cross-platform mobile development tools. In 16th International Conference on Intelligence in Next Generation Networks, ICIN 2012, Berlin, Germany, October 8-11, 2012, pages 179–186, 2012. doi: 10.1109/ICIN.2012.6376023. URL <http://dx.doi.org/10.1109/ICIN.2012.6376023>.

- [Parr(2000)] Terence Parr. String template, 2000. URL <http://www.stringtemplate.org>.
- [Parr(2007)] Terence Parr. The definitive antlr reference: building domain-specific languages. 2007.
- [Reda(2009)] Renas Reda. Robotium, 2009. URL <http://www.robotium.com/>.
- [Rösler et al.(2014)Rösler, Nitze, and Schmietendorf] Florian Rösler, André Nitze, and Andreas Schmietendorf. Towards a mobile application performance benchmark. In ICIW 2014, The Ninth International Conference on Internet and Web Applications and Services, pages 55–59, 2014.
- [Steinberg et al.(2003)Steinberg, Budinsky, Merks, and Paternostro] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. EMF: eclipse modeling framework. Pearson Education, 2003.
- [Systems(2009)] Adobe Systems. Phonegap, <http://www.phonegap.com/>, 2009. URL <http://www.phonegap.com/>.
- [Wolf and HUFFSTADT(2013)] Florian Wolf and KARSTEN HUFFSTADT. Mobile enterprise application development-a cross-platform framework. FHWS Science Journal, page 33, 2013.
- [Yang et al.(2013)Yang, Prasad, and Xie] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, pages 250–265, 2013. doi: 10.1007/978-3-642-37057-1. URL <http://dx.doi.org/10.1007/978-3-642-37057-1>.