Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

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

Mohamad Jaber[1], **Yliès Falcone**[2], **Kinan Dak-Al-Bab**[1], **John Abou-Jaoudeh**[1], **Mostafa El-Katerji**[1]

[1] American University of Beirut, Beirut, Lebanon
e-mail: `mj54@aub.edu.lb`, `{kmd14,jia03,mme85}@mail.aub.edu`

[2] Univ. Grenoble-Alpes, Inria, Laboratoire d'Informatique de Grenoble, Grenoble, France
e-mail: `Ylies.Falcone@imag.fr`

The date of receipt and acceptance will be inserted by the editor

Abstract. Developing mobile applications remains difficult, time consuming, and error-prone, in spite of the number of existing platforms and tools. In this paper, we develop MoDroid, a high-level modeling language to ease the development of Android applications. MoDroid allows developing models representing the core of applications. MoDroid provides Android programmers with the following advantages:(1) Models are built using high-level primitives that abstract away several implementation details; (2) It allows the definition of interfaces between models to automatically compose them; (3) A native Android application can be automatically generated along with the required permissions definition; (4) It supports efficient testing execution that operates on models. MoDroid is fully implemented and was used to develop several non-trivial Android applications.

1 Introduction

Android is the most popular platform for mobile devices, with over 84% of market share at the end of 2014. Yet, creating a correct and efficient Android application remains a difficult endeavor for several reasons that can be categorized under design or testing issues.

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 exceptions 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 [5].

Issues when testing an Android application. On the other hand, 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.

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.

Meta-Model and Android Java models. We implement a Meta-Model along with several auxiliary modules in MoDroid to circumvent the aforementioned problems. Our implementation consists of a set of modules (Java classes) that represent Graphical User Interfaces (e.g., Text Views, Buttons) and their respective handlers (e.g., on click, on edit). The modules/classes can be instantiated into multiple instances. Instances are placed into a tree hierarchy using a well-defined API. The root element in this hierarchy is the model (i.e., Application) object. This hierarchy can be automatically converted (code generation and translation) to a Native Android Application and is referred to as an Android Java model. More precisely, 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 automatically discovers the required permissions of an application.
3. A code generator 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 testing framework for the model that allows to easily write test cases using high-level primitives and to efficiently execute them.

Figure 1 shows a high-level map of the development design-flow which is based on MoDroid. 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 functionality. Consequently, it is designed with backward

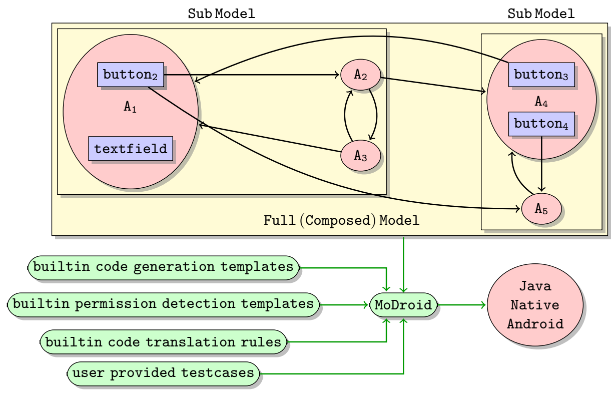

Fig. 1. High-level description MoDroid design flow.

compatibility in mind so that developers can write native Android code within the model to use features currently not covered within the Meta-Model.

Paper organization. The rest of this paper is structured as follows. Section 2 presents the Meta-Model. The following sections present the components associated to the Meta-Model: model composition is presented in Sec. 3; and automatic permissions detection is presented in Sec. 4; a testing framework is presented in Sec. 5; and automatic code generation (from high-level model to native Android) is presented in Sec. 6. Sections 7 and 8 describes MoDroid, a full implementation of our framework and some benchmarks. Section 9 discusses related work. Section 10 draws some conclusions and perspectives.

2 The Android Meta-Model

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 paper.

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.

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.

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. 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.

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).

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 2 depicts some of the basic elements of the Meta-Model. Other elements exist in the Meta-Model but are not shown for the sake of simplicity (e.g., LibRelativeLayout, LibCheckBox, LibSpinner). Moreover, the Meta-Model includes other internal classes such as generators, parsers, translators. Hereafter, we show a step-by-step 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 1 shows a code snippet of the code of BMI module.

2.1 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 predefined shortcuts (i.e., syntactic sugar). The code within the handlers can use functionality from the Meta-Model or can 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 filed gets modified, one common functionality is to go to another activity. For a given view, one specifies its handler method by calling setOnClickHandler. 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 setOnClickHandler. It is worth noting that the handler is used as a code wrapper. The code inside the handler is automatically parsed and translated to native Android code. That is, references to the model are replaced by their counter-parts in Android. Note that, since the handlers are code wrapper, parts of the method's header are ignored during generation (e.g., access modifiers, static).

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 setOnClickHandler.

Listing 2 shows the code of the button from the first activity where its handler computes the BMI value and send 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.

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 shows a snapshot of the code that sets some of the views of the second activity. It sets the 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.

2.2 Resource Management

One of the most effort consuming task 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.

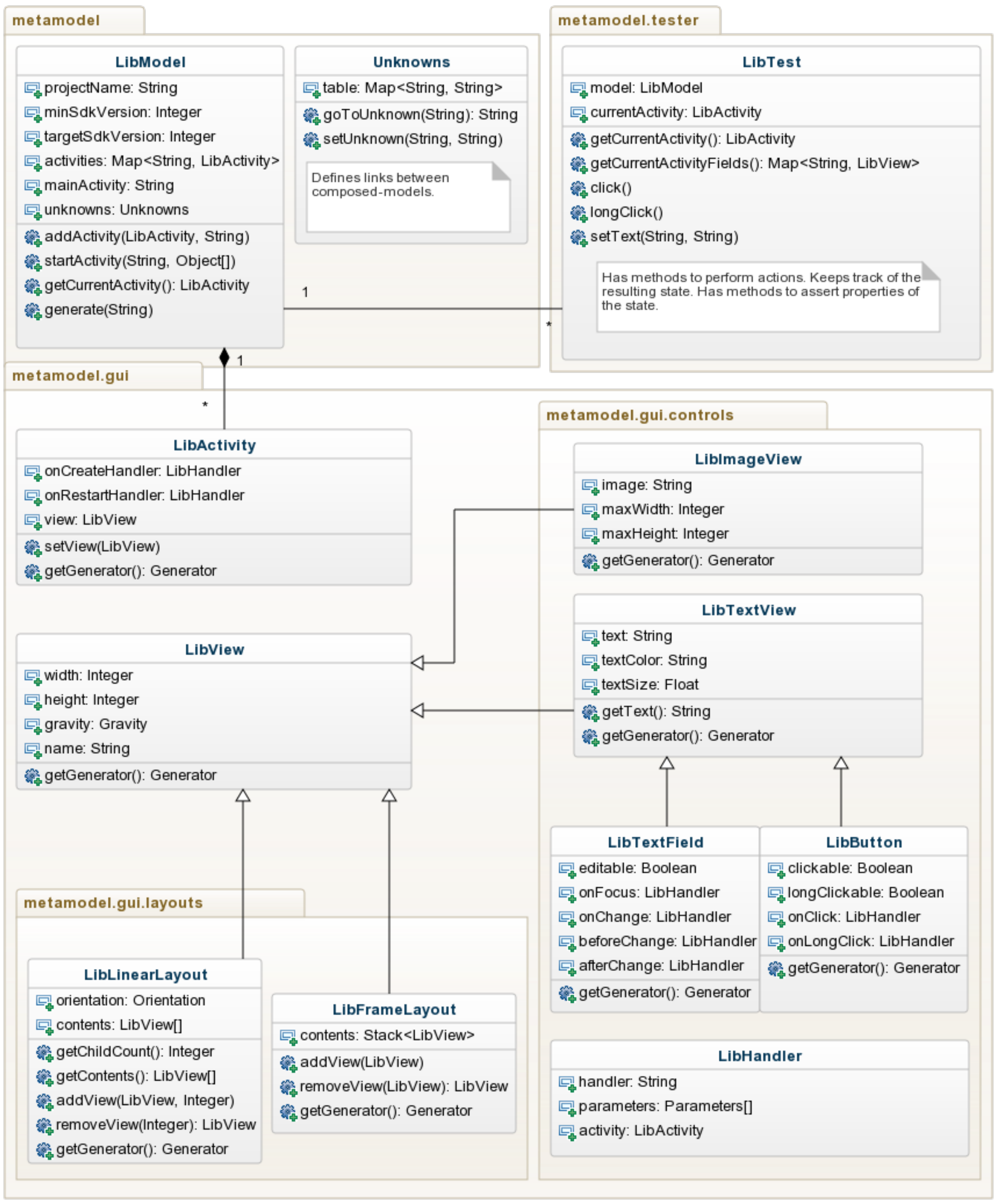

Fig. 2. Basic elements of the Meta-Model

Listing 1. Code snippet of BMI module.

```java
LibModel bmiModel = new LibModel("bmiModel", "health.app", "John");
LibActivity userInputActivity = new LibActivity();
LibActivity resultActivity = new LibActivity();
bmiModel.addActivity(userInputActivity, "userInputActivity");
bmiModel.addActivity(resultActivity, "resultActivity");
setUserInputActivityLayout( userInputActivityLayout );
setResultActivityLayout( resultActivityLayout );
...
```

Listing 2. Example of a handler with data transfer.

```java
calculateButton.setOnClickHandler("Handler:health.BMI.calculate", height, weight);

// package health
// class BMI
public void calculate(LibView ht, LibView wt) {
   if(!ht.getText().equals("") && !wt.getText().equals("")) {
      double val = computeBMI (ht, wt);
      LibModel.startActivity("resultActivity", val);
   }
   else {
      Vibrator v = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
      if(v.hasVibrator()) v.vibrate(500);
   }
}
```

Listing 3. Example of shortcut handler and data access.

```java
bmiValueText.setText("@param_0");
...
goBackButton.setOnClickHandler("GoToActivity:userInputActivity");
```

For example, to use an image, the developer only needs to add the path of the image/icon to be used. Listing 4 shows an example that specifies the icon of an application, displays an image, and create a button with an image displayed.

3 Projects Composition

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. 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 occur 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.

Principles. Given n models $m_1, m_2, \ldots, m_n$, where m_i consists of $a^i_1, a^i_2, \ldots, a^i_{I_i}$ 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 Sec. 7). Function configuration is defined by the user through a configuration file. A configuration file is of the form depicted in Listing 5. 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^i_1, id^i_2 (interface$(m_i) = \{id^i_1, id^i_2\}$). Let a^j_k be an activity of model m_j, one can have configuration$(id^i_1) = a^j_k$, which means that identifier id^i_1 of model m_i is mapped to activity a^k_j of model m_j.

Example. Figure 3 is an example of two partial models $\texttt{M}_1$ and $\texttt{M}_2$. The handler of button $\texttt{button}_2$, a handler of activity $\texttt{A}_2$ and the handler of button $\texttt{button}_3$ redirect to symbolic activities though interfaces $\texttt{I}_1$, $\texttt{I}_2$ and $\texttt{I}_3$, respectively. The configuration file connects $\texttt{I}_1$, $\texttt{I}_2$ and $\texttt{I}_3$ to activities $\texttt{A}_5$, $\texttt{A}_4$ and $\texttt{A}_1$, respectively.

Listing 6 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`.

Finally, models can be composed to build the final project by using `LibModel`'s constructors that takes a

Listing 4. Example of resource management.

```
  ...
  model.setIcon("images/application.jpg"); // sets the application icon
  // Create a label to display the given image.
  LibImageView imageView = new LibImageView("images/image.jpg", ...);

  // Create a button with an image displayed on it.
  LibImageButton imageButton = new LibImageButton("images/button.jpg", ...);
```

Listing 5. General shape of a configuration file.

```
<New Project Name>
<New Project Package>
<New Project Author>
<Model>.<Activity>; //indicates the main activity of the composed project
<Model>.<Unknown ID> -> <Model>.<Activity Name>; // mapping
<Model>.<Unknown ID> -> <Model>.<Activity Name>; // mapping
<Model>.<Unknown ID> -> <Model>.<Activity Name>; // mapping
...
```

Listing 6. Example of unknown shortcut handler.

```
menuPlannerButton.setOnClickHandler("GoToActivity:Unknowns(menuPlannerInterface)");
```

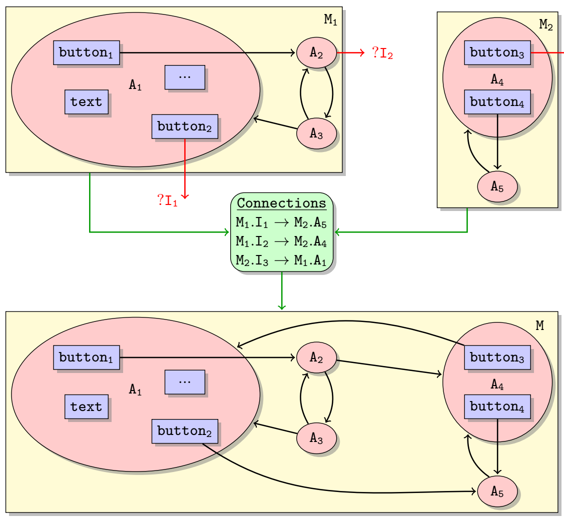

Fig. 3. Example of models composition.

configuration file and a set of models. The composition of BMI and Menu Planner modules is depicted in Listing 7.

Listing 8 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.

Note that, a set of models can be composed successively to build the final model. Listing 9 shows an example of successively composing three models.

Although mobile applications almost certainly harbors 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.

4 Permission Auto-detection and Generation

Native Android development requires to list and declare in a configuration file the used permissions. This allows users to control the applications' usage of any "dangerous permissions". Starting from Android 23, a new permission scheme was introduced, which allows applications to ask for permissions at runtime. The application is responsible for checking whether it has the required permissions before executing. If the application does not have the required permissions, the user is asked to grant these permissions at runtime. The user can revoke permissions from an application at any time. The previous permission scheme would list all needed permissions to the user at installation time. Both approaches require permissions to be listed in the manifest file so as to allow applications to have a correct flow in both schemes. The scheme in use is determined by the target SDK of the application as well as the running Android version.

Listing 7. Composition of BMI and Menu Planner modules.

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

Listing 8. Configuration file connecting BMI and Menu Planner models.

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

Listing 9. Successive composition of models.

```
LibModel model12 = new LibModel("config1.txt", model1, model2);
LibModel model123 = new LibModel("config2.txt", model12, model3);
```

Listing 10. 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)
7         getSystemService(Context.VIBRATOR_SERVICE);
8
9     if(v.hasVibrator()) {
10       Toast.makeText(this, text, duration).show();
11    }
12 }
```

In both schemes, manually managing permissions is time consuming and error-prone. It often entails several compilation attempts of the application to narrow the proper set of required permissions. Consequently, most of the developers add permissions more than it is needed which contradicts with the users' preferences and privacy. For 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. Listing10 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 require a manual reconsideration of the required permissions. In our case, APIs to access devise-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 requires a permission. For instance, object initialization (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 11. The template depicted in Listing 11 defines all the API calls shown in 12 that require permission `PERMISSION_1`:

For example, the template for permission `Android.permissions.VIBRATE` is depicted in Listing13. From the template of permission `Android.permissions.VIBRATE`, we can deduce that permission `Android.permissions.VIBRATE` is

Listing 11. 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>
```

Listing 12. API calls requiring permission PERMISSION_1.

```

(Class_1).method_1(param1);

(Class_1).method_2();

(Class_1).method_3(param2, param3);

(Class_1).method_3(param4);

(Class_2).method_4(...);

(Class_2).method_5(...);

Class_3 var = new Class_3(...);
```

Listing 13. 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>
```

required whenever one of the lines of code in Listing14 is detected.

Listing 14. API calls requiring permission `Android.permissions.VIBRATE`.

```
// v is an object of type Vibrator
// E.g., Vibrator v = (Vibrator)
//   getSystemService(Context.VIBRATOR_SERVICE);


// vibrate(long milliseconds) method
v.vibrate(500);

// vibrate(long[] pattern, int repeat) method
v.vibrate({{12}, {23}, {12}}, 50);

// cancel() method
v.cancel();
```

Since the user can revoke permissions from the application, the code of the application must check that the application holds the permission required by the code it executes. This is a tedious process which can be easily missed by developers. When it is discovered that certain lines of code require certain permissions (based on the same technique as above), we automatically surround them with some primitives to ensure that the permissions are granted when executing. This is simply done by calling `ContextCompat.checkSelfPermission`, and providing it with the needed permission.

5 Testing The Model

In order to integrate efficient 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 activities scope variables and global variables. We implement all the functionality to perform operations on a given model. For example: (1) modify or get the value of a view; (2) perform click/event. In order to perform a click, we use Java reflection to execute the handler of a corresponding view (e.g., button). Performing operations modify the state of the model accordingly.

The 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.

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.

Listing 15 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 give an activity 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.

Note that, if one performs an operation on a view that does not exist in the current activity, an exception is thrown.

6 Code Generation

Finally, 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 object. The generated code preserves the order of statements and comments. This allows to easily integrate other functionality to the generated code.

The code generation consists of the following phases:

1. For each component of the Model (e.g., `LibTextView` instance), an equivalent Android component is generated (e.g., `TextView` instance).The attributes and data set inside the component (e.g., text, width and colors) are transferred to its Android counter part. Resources attached to these components (e.g, images) are moved and packaged to the corresponding destination code.
2. For each component, the attached handlers code is read and parsed. Each line of code is analyzed, and actions are taken accordingly:
   - If the line of code requires a specific permission, this permission is automatically added to the manifest file.
   - If the line of code contains a reference to a component of the model, this line of code is translated to one or many native Android lines of code by applying the corresponding translation rules. Translation rules are predefined symbolic rules inside of MoDroid. A match between a rule and a statement in a handler is determined by comparing a mixture of method names, class names, parameter count, and data types.
   - If native code is encountered, it is copied as is to the destination code (sometimes variable names are modified to ensure consistency).
3. The generated Android components are put together in a hierarchy matching that of the model. This hierarchy matches the GUI hierarchy which is visible in the application when it is running.

This approach offers several advantages:

1. The model and the handlers are both parsed and stored in memory, which allows for easy code manipulation and static analysis.
2. The resulting native Android code has no reference to MoDroid and its components. Manually modifying or extending the code (if a developer wishes) is straight forward for an experienced Android developer.
3. The approach entails no slow-down since the translation and generation is entirely done at compile-time (local or remote). The actual running application will contain usual simple native Android calls without any re-routes or abstractions.
4. Native Android code can be directly written inside handlers.
5. Dynamic code analysis tools, testing tools, and any other helping tools can be used with the generated code to facilitate development, and allowing MoDroid to be cross-compatible with all common tools used in Android Development.

If the generation of the code encountered a problem with a referential integrity of the model, an appropriate exception will be thrown. The exception contains details on the error and its origin (e.g., handlers that do not exist, mismatches argument types).

6.1 Cloud-based Compilation and Testing

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 package file (`.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)

Listing 15. Example of test cases.

```java
@Test
public void testcase1(LibModel bmiModel) {
   try {
      LibTest test = new LibTest(bmiModel);
      test.setText("height", "175");
      assertEquals("Incorrect Height", "175", test.getText("height"));

      test.setText("weight", "70");
      assertEquals("Incorrect Weight", "70", test.getText("weight"));

      test.click("calculateButton");

      assertEquals("Incorrect Activity", "resultActivity", test.getCurrentActivityName());
      assertEquals("Incorrect Value", "22.9", test.getText("value"));
   } catch (ElementNotFoundException e) {
      fail("Element Not Found: " + e);
   }
}
```

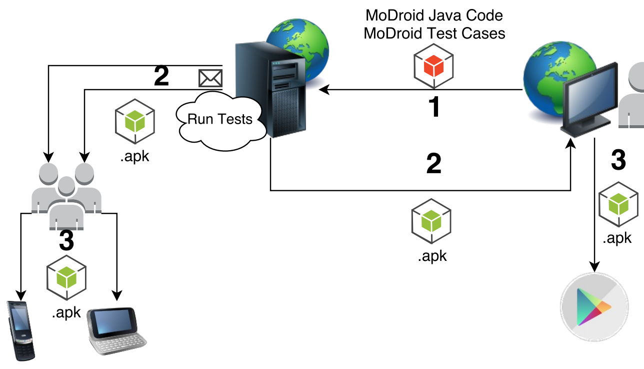

Fig. 4. Cloud-based Compilation and Testing

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

7 Tool-set - MoDroid

MoDroid[1] implements the Meta-Model and its supported tools: models composition, 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.

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 implementation of that interface:

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

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

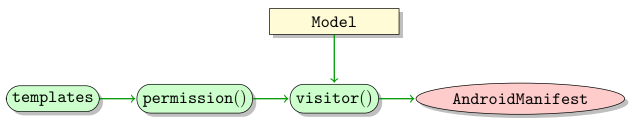

Fig. 5. Permission detection flow

3. Implementation to make the model executable by performing operations on a view (e.g., `LibText`) that are used by the testing module.

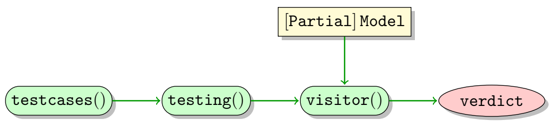

Fig. 6. Testing flow

4. Implementation to generate equivalent native Android code (along with its resources, manifest configuration file, etc.) from an Android model. Code generation module uses `antlr` for parsing handlers and template engine library `StringTemplate` [17] for generating native Java Android from an Android model

[1] `http://ujf-aub.bitbucket.org/modroid/`

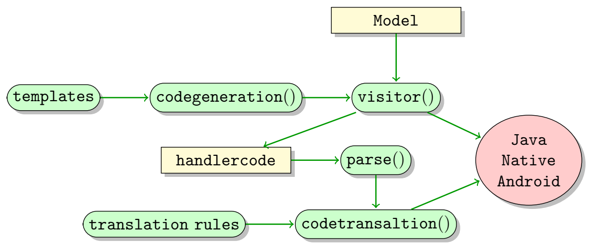

Fig. 7. Code generation flow

First, 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.

Second, 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. Finally, a native Java Android code is generated including all the permissions that are required by the core of the application. Developers may edit the generated code to add any extra functionality.

8 Experimental Results

We have developed several applications (Breadcrumb Viewer, Guessing Game, Scientific Calculator, and Volleyball Statistics) using both native Java Android and MoDroid. Both versions of the code have the same design and perform exactly the same functions. Table 1 compares the number of lines of code 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 w.r.t. writing native Java Android. We notice an overhead of ca. 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.

Moreover, we have conducted other benchmarks to compare the performance of our testing framework and the following tools that are currently widely used: Robolectric [14], Robotium [18], and Espresso [12] 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.

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

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.

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

Table 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.

9 Related Work

We propose to overview and compare work related to the approach proposed in this paper. We mainly distinguish two kinds of related approaches:

– Development of Android applications.
– Testing of Android applications.

9.1 Android Mobile Development

This paper advocates the use of modeling to improve the development of Android applications. 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) [20] is a powerful modeling tool based on

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 1. Code comparison.

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 2. Testing Time (in seconds).

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 3. Testing Time (in seconds).

two meta-models 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 [8], another tool from Eclipse, is a textual syntax for Ecore. Both EMF and Xcode 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. While EMF is very helpful in constructing data model that is independent from application logic and user interface, Android applications usually do not involve heavy data processing, and are concerned mainly with user interface and device-based features such as GPS and Camera. EMF can offer minimal support for such application since its generated models only cover basic Java syntax and do not support Android API. Additionally, EMF provides a general and modular code generation (e.g., several hierarchy of classes with several predefined methods), and hence this may deteriorate the performance of the generated code, which is critical for mobile applications.

Mobile development frameworks are usually categorized into native, cross-platform, and web based. 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 web based allows for the fastest development. We compare our approach with some of the frameworks in those categories:

- **native:** App Inventor 2 [15] 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 the application itself since users cannot write their own code, and are only limited to what is provided by the GUI.
- **hybrid:** PhoneGap [21] and Cordova [2] are two commonly used cross platform mobile development frameworks [16]. They allow the developer to generate mobile applications that work on almost all devices by using HTML, CSS, 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. Furthermore, performance issues were reported due to the lack of hardware CSS acceleration of Android [22].
- **web-based:** jQuery mobile [13] 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.

Two main disadvantages arise when using web based frameworks: poor performance [19], and loosing the ability to use smartphone features.

Finally, none of the above Android development frameworks allows for the composition and decomposition of applications. MoDroid allows for this, as shown in Section 3. Moreover, it allows for permission auto-detection and generation as specified in Section 4. 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 [7] [4].

Moreover, although MoDroid provides an abstract API that alleviates the burden placed on developers, it also allows the developer to write native code into the model (e.g., through handlers). This guarantees that MoDroid is equally expressive, and has full access to the device features.

MoDroid automatically generates native Android code. Hence, this avoids the slow down associated with using Javascript and CSS. Moreover, the generated Android code follows immediately from the code written by developers, since the translation simply replaces references to the model with equivalent references to the Android API. This guarantees the similarity at runtime of MoDroid-generated applications and natively written applications.

9.2 Modular Development of Android Applications

MoDroid provides the ability to develop separate models, and then compose them into one model using a configuration file. Each of these models is a complete unit. That is, a model may contain processing data, user interface, network interactions, and any other functionality. Any aspect/task of an application can be developed as a single/separate model.

Native Android provides services out of the box, which allows the developer to isolate and modularize certain low-running operations in the background. Services can be either part of the code-basis, or they can be provided by a third party library (with a corresponding Android Interface Definition Language (AIDL) file). However, services cannot contain GUI components. Moreover, these services have a special life-cycle: (1) start by an application component, (2) run in the background. They continue executing even if the user leaves the application. Services can be bound to a component, if so the service will be destroyed upon unbinding. Although services are very useful, they can only run tasks in the background (e.g., music, network transaction). They cannot be used for decomposing an application into separate identical modules.

Zneika et al [25] proposed a "modular and lightweight" service oriented approach for service management. The main objective of [25] is to avoid the performance slow-downs that are incurred when using AIDL based solutions. In general, slow-downs are due to the usage of multiple layers of middleware between the application and the external service. For this, the solution in[25] removes the middle layers by using modules written in ready-to-execute Dalvik code. Nonetheless, although this approach may speed up the invocation of services, it also lacks support for GUI. The used services are executed at the level of the operating system in the background, and have an identical scope to that of out-of-the-box Android Services.

Compositions of models using MoDroid allows to decompose applications into small tasks/units, each of which can be fully dealt with by a dedicated model. Then, models can be composed together at generation time without any additional runtime cost, since the invocation of such model is translated to a simple activity call in the generated code. Models may also be tested separately.

9.3 Android User Interface (UI) Design

MoDroid allows the user to design the UI of the application programmatically using the different classes it provides. This process is similar to designing the UI programmatically using the Android API. Other tools such as Android Studio comes with a built-in Layout Editor [11], which allows users to drag and drop views into other layouts, as well as visually control their properties (like width/height). The editor manipulates the XML describing the UI according the user's interaction. Although the Layout Editor is very useful, it is restricted to creating the UI and has little effect over functionality or code design. In the future work, we consider extending MoDroid by adding a rigorous and expressive graphical modeling framework that benefits from our Meta-Model and simplifies the development process.

Similarly, XCode, iOS tool-set, provides a visual way to control the end-user path through the application (between different screens) using Storyboards [3]. Storyboard are used to specify scenes (an area of the screen), segues (transitions between scenes), relationships between containers and scenes, as well as controls that trigger segues. Nonetheless, it is specific for iOS applications and not for Android.

9.4 Testing Android Applications

On the other hand, testing of Android applications become more challenging. In general, Android testing tools can be divided into two main categories: GUI based testing and non-GUI based testing.

GUI based testing: This category requires testing on an emulator or on a real Android device. Google present several tools some of which fall under this category. First is Instrumentation [9], 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 [12]. Espresso is built over Instrumentation and its main goal is to simplify testing techniques.

Another commonly used tool is Robotium [18]. 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 [10], Android GUITAR [1] and ORBIT [23].

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 8). 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 devises (e.g., camera, sensors).

Non-GUI based testing: Robolectric [14] 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 of several other libraries. In addition, developing test cases is complicated. For instance, Listing 16 is a sample code to access the value of a `TextView` using Robolectric. Our framework falls under the category of non-GUI based testing. We target ease of configuration, simplicity and performance.

Listing 16. Sample code to access the value of a `TextView` using Robolectric.

```
ActivityClassName activity = Robolectric.
        buildActivity(ActivityClassName.class).
        create().start().visible().get();

TextView results = (TextView) activity2.
        findViewById( viewID );

results.getText();
```

Running MoDroid tests only requires to import the MoDroid JAR file as a library to the project. MoDroid does not require any additional configuration for testing, which makes it easy to run tests in the cloud. Moreover, test cases can be attached to hooks so that they are automatically performed whenever a code modification happens, or when code is pushed to a remote repository (e.g., git hooks).

9.5 Code Analysis of Android Applications

Dynamic Program Analysis of Java programs have been studied in many cases. Several libraries and tools exist for tracing and profiling Java programs. However, the proposed tools do not extend directly to Android applications. In general, dynamic analysis of Java often entails instrumentation of low-level Java byte code. This raises many problems when attempting such an analysis on Android code because of the differences between the JVM and the Dalvik Virtual Machine used on Android devices. Different components of the same application may be executed using different DVMs. Communication between these components is done through IPC. DVM also handles loading and unloading of classes differently than JVM. In [24], the authors proposed a platform that enables such analysis on Android by modifying the DVM to provide features such as object tagging, hooks in the garbage collection. The modifications are encapsulated in interfaces that can be ported into new Android Runtime (ART). However, this adds a communication overhead. Moreover, object tagging has proved to be a bottleneck that is stressed by their implementation, which slows down the execution. JaCoCo [6] is a library for code coverage for Java. It identifies the lines of code, as well as classes and methods that were used during execution. JaCoCo can be used with Android code provided a proper configuration of the corresponding `Gradle` plugin of the application.

Dynamic Analysis Tools can be used with MoDroid in two different phases:

1. *Before generation:* Using MoDroid allows to run test cases on any machine equipped with JVM (provided no native code was written directly into the handlers). Running the MoDroid test cases involves running the required handlers and keeping track of the state of the application within the model. DVM will not be involved with the execution at this stage. Thus, test cases can be combined with any dynamic program analysis tool for Java. For instance, any code coverage tool can be used while running a MoDroid test. The lines executed in handlers will be recorded and highlighted by the IDE (if the coverage tool has IDE integration). This provides an advantage to the developers who want to use analysis tools that support Java without extending the support to Android and DVM. Moreover, it avoids the overhead and slowness of running such analysis on the DVM.

2. *After generation:* The final product of a MoDroid model is a working Android application. The application can be passed to dynamic analysis tools and processed as any regular Android application. Developers can use the feedback of these tools to modify either the initial model or the generated code.

During the code generation-translation scheme, the code within the handlers of the model is parsed and then analyzed for any references to the model and other features such as permission detection. This makes it easy, rigor and feasible to add extra analysis to MoDroid. Moreover, it allows for high-level instrumentation through injecting of high-level Java snippets in desired key points in the code, which can help simplify dynamic analysis on DVM. These modifications are easy to integrate into MoDroid code.

9.6 Compatibility

Other testing tools (GUI and non-GUI based) can be used in combination with MoDroid, since the generated application is just a regular Android application. Developers can run any testing or code analysis tool they desire on the resulting application. Results can be traced back to the original model by identifying which handler the related pieces of code belong to. This can be easily identified through the containing method name. Narrowing the scope down to a single handler makes it easy for the developer to compare the generated code to the original (since handlers are small single-purpose pieces of code). Automatically tracing specific lines of code from the generated model back to the model is possible as long as some information is kept from the generation phase.

10 Conclusion and Future Work

This paper proposes a new way to develop Android applications. It proposes a compromise between expressiveness and ease of development at the price of slightly reduced 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.

Native Android code can be directly written inside the handlers. Native code is carried out to the generated application and it can be modified and analyzed if need be.

Moreover, our framework introduces several interesting features for developers: decomposition of applications for parallel development, automatic detection of permissions and generation of Manifest, and efficient model-based testing of applications.

For future work, we plan to consider several directions:

1. *Richness of MoDroid Constructs:* We plan to add several features to the road-map of MoDroid. First, we plan to add emulators for hardware components such as the GPS and camera. This will allow the user to e.g., pre-define GPS locations to be passed to the application. Moreover, we plan to extend MoDroid to support a high-level description of multi-tasking, services, broadcast receivers, etc.
2. *Generation, translation, and analysis:* We plan to extend the code generation and translation scheme to include more analysis of the code. This will facilitate the dynamic analysis of MoDroid code. More precisely, we consider integrating a generic code analysis that will be parameterized with analysis models. We also consider to implement automatic code traceability from the generated application back to the model. This can be achieved by keeping a map between key constructs in the generated code and the parts of the model it was generated by. Then computing the lines of code offset when needed.
3. *Graphical Modeling Tool:* We plan to extend MoDroid by adding a graphical modeling tool from which models can be automatically generated.

References

1. 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.
2. Apache. Cordova, `http://cordova.apache.org/`, 2011.
3. Apple. Storyboard, `https://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html`, 2013.
4. 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.
5. 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.
6. EclEmma. Jacoco java code coverage library.
7. Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *SIGSOFT FSE*, 2014.
8. 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.
9. Google. Testing instrumentation, 2007.
10. Google. Application exerciser monkey, 2010.
11. Google. Layout editor, `https://developer.android.com/studio/write/layout-editor.html`, 2011.

12. Google. Espresso, 2013.
13. jQuery Team. Jquery mobile, `http://www.jquerymobile.com/`, 2010.
14. Pivotal Labs. Robolectric, 2010.
15. Edward Mitchell. *App Inventor 2: Tutorial: The fast and easy way to create Android apps*, volume 1. Edward Mitchell, 2014.
16. 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.
17. Terence Parr. String template, 2000.
18. Renas Reda. Robotium, 2009.
19. 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.
20. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2003.
21. Adobe Systems. Phonegap, `http://www.phonegap.com/`, 2009.
22. Florian Wolf and KARSTEN HUFFSTADT. Mobile enterprise application development-a cross-platform framework. *FHWS Science Journal*, page 33, 2013.
23. 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.
24. Y. Zheng, S. Kell, L. Bulej, H. Sun, and W. Binder. Comprehensive multi-platform dynamic program analysis for java and android. *IEEE Software*, PP(99):1–1, 2015.
25. Mussab Zneika, Hasan Loulou, Fatiha Houacine, and Samia Bouzefrane. Towards a modular and lightweight model for android development platforms. *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 2129–2132, 2013.

11 Appendix

We show a small example to showcase syntax used in MoDroid. Moreover, we show parts of the generated application structure. We emphasize the simplicity and readability of generated code. This ensures that developers have the ability to modify and edit the generated code if they want to. Line-by-line traceability of the generated handlers to the original code is also clear. We also show the generated manifest and the detected permission as well.

Listing 18 and 19 contains the full code of the MoDroid model. The model represents a simple 2-activities application. The first activity contains a simple login screen, upon submitting the application moves to the second activity, and displays the username used. Listing 19 also contains the generated code for the handlers.

The generated application has the following structure:

```
|-- Simple Login Model
    |-- AndroidManifest.xml
    |-- res
    |   |-- drawable
    |   |   |-- ic_launcher.png
    |   |-- values
    |       |-- strings.xml
    |       |-- styles.xml
    `-- src
        |-- LoginActivity.java
        |-- SecondActivity.java
```

Listing 17 shows the generated manifest that contains the required permissions and information about the activities of the application. Clearly, developing this application using native Android is more complex and more time consuming than using MoDroid. This is mainly due to the following: (1) native Android requires more code (up to four times more) than MoDroid; (2) we should manually write configuration file with the proper permissions which is automatically generated in case of MoDroid; (3) testing using native android requires extra configurations and running tests will take place on emulators. On the contrary, testing using MoDroid does not require any extra configuration. Moreover, running tests will be directly simulated on the underlying model which is much more efficient that running them on emulators.

Listing 17. Automatically Generated Manifest File.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
   "http://schemas.android.com/apk/res/android"
   package="foo.bar"
   android:versionCode="1"
   android:versionName="1.0" >

<uses-sdk
   android:minSdkVersion="8"
   android:targetSdkVersion="19"/>

<!-- REQUIRED APPLICATION DETECTED -->
<uses-permission
   android:name="android.permissions.VIBRATE" />

<application
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name"
   android:theme="@style/AppTheme">

   <activity
      android:name=".LoginActivity"
      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=".SecondActivity"
      android:label="SecondActivity">
   </activity>

</application>
</manifest>
```

Listing 18. MoDroid Model Code

```java
import edu.lb.aub.android.metamodel.LibModel;
...

public class Application {

public static void main(String[] args) throws FileNotFoundException, IOException {
  LibModel testProject = new LibModel("Simple Login Model", "foo.bar", "Testers");

  LibActivity loginActivity = new LibActivity(); //First Activity is a login window
  LibActivity displayActivity = new LibActivity(); //Second Activity displays username
  buildLogingActivity(loginActivity);
  buildDisplayActivity(displayActivity);

  testProject.addActivity(loginActivity, "LoginActivity");
  testProject.addActivity(displayActivity, "DisplayActivity");
  testProject.setMainActivity("LoginActivity"); // sets the main/starting activity
  testProject.generate("gen-test/"); // generate code into the given directory
}

public static void buildLoginActivity(LibActivity loginActivity) {
  LibLinearLayout layout = new LibLinearLayout();
  layout.setOrientation(LibLinearLayout.ORIENTATION.vertical); // display content vertically

  layout.setWidth(LibView.MATCH_PARENT); // 100% of parent's width
  layout.setHeight(LibView.WRAP_CONTENT); // height is that of contents
  layout.setGravity(LibView.GRAVITY.center_horizontal); // contents are centered

  LibTextField usernameField = new LibTextField(200, 30); // text field with width and height
  layout.addView(usernameField); // add view to layout

  LibTextField passwordField = new LibTextField(200, 30);
  layout.addView(passwordField);

  LibButton loginButton = new LibButton(60, 30, "Login"); // button with width, height, and Text
  loginButton.setMarginBottom(20);
  layout.addView(loginButton);

  LibButton exitButton = new LibButton(60, 30, "Exit");
  layout.addView(exitButton);

  //Assign on click handlers to buttons, login handler has two parameters
  //package name : test. class name : Application. method name : loginHandler.
  loginButton.setOnClickHandler("Handler:test.Application.loginHandler", usernameField, passwordField);
  exitButton.setOnClickHandler("Handler:test.Application.exitHandler"); //no parameters

  loginActivity.setView(layout); // set the activity to display layout
}

public static void buildDisplayActivity(LibActivity displayActivity) {
  LibLinearLayout layout = new LibLinearLayout();
  layout = new LibLinearLayout();
  layout.setOrientation(LibLinearLayout.ORIENTATION.vertical); // display content vertically

  // Gets the first parameter sent from previous activity and display it.
  LibTextView textView = new LibTextView(LibView.MATCH_PARENT, LibView.WRAP_CONTENT, "@param_0");
  layout.addView(textView);
  displayActivity.setView(layout); // set the activity to display layout
}
```

Listing 19. MoDroid Handlers cont.

```java
public void loginHandler(LibTextField usernameField, LibTextField passwordField) {
   String username = usernameField.getText(); //Reference to the model, will be translated.
   String password = passwordField.getText(); //Reference to the model, will be translated.

   if(username.length() < 5 || password.length() < 5) return; // validate lengths of input.

   // do something with username and password

   // Go to the other activity, send username and password as parameters.
   LibModel.startActivity("DisplayActivity", username, password); //Reference to the model.
}

public void exitHandler() {
   // Native code, will be moved as is to the handler.
   Vibrator v = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);

   // Requires Permission, will be added automatically.
   if(v.hasVibrator()) v.vibrate(500);

   System.exit(0); // Not a reference to the model.
}
```