Effective Testing of Android Apps Using Extended IFML Models

Minxue Pan[a,c], Yifei Lu[a], Yu Pei[b], Tian Zhang[a,c], Juan Zhai[a], Xuandong Li[a]

[a]*State Key Laboratory for Novel Software Technology, Nanjing University*
[b]*Department of Computing, The Hong Kong Polytechnic University*
[c]*Corresponding authors*

Abstract

The last decade has seen a vast proliferation of mobile apps. To improve the reliability of such apps, various techniques have been developed to automatically generate tests for them. While such techniques have been proven to be useful in producing test suites that achieve significant levels of code coverage, there is still enormous demand for techniques that effectively generate tests to exercise more code and detect more bugs of apps.

We propose in this paper the Adamant approach to automated Android app testing. Adamant utilizes models that incorporate valuable human knowledge about the behaviours of the app under consideration to guide effective test generation, and the models are encoded in an extended version of the Interaction Flow Modeling Language (IFML).

In an experimental evaluation on 10 open source Android apps, Adamant generated over 130 test actions per minute, achieved around 68% code coverage, and exposed 8 real bugs, significantly outperforming other test generation tools like Monkey, AndroidRipper, and Gator in terms of code covered and bugs detected.

Keywords: Interaction Flow Modeling Language, Android apps, Model-based testing

1. Introduction

The past few years have seen a rapid growth in the popularity of mobile devices and applications running on them, or mobile apps [1]. To ensure the reliability of mobile apps, developers conduct various quality assurance activities, among which testing is the most frequently performed [2, 3, 4]. For testing to be effective, tests of good quality are essential, but manual construction of those tests can be tedious and highly time consuming, leading to increased costs for mobile app testing.

In view of that, researchers developed many techniques and tools over the years to automatically generate tests for mobile apps. Most of such works target the Android platform, mainly due to its open-source nature and the fact that it has the largest share of the mobile market [5]. For instance, Monkey [6] is a representative of the state-of-the-art Android test generation techniques. Monkey implements a random strategy to automatically generate test scripts, and it is more effective than most other Android test generation tools that are publicly available [5]. Relying solely on computer automation, Monkey is good at having simple interactions with the app under testing. It, however, lacks a good knowledge of the app and has limited power in exercising important and complex functionalities of the app. As a result, code coverage achieved by test scripts produced by Monkey is often insufficient: Less than 50% of the code was covered in an experiment on 68 open-source apps [5], and lower code coverage was reported in another experiment with an industrial-level app [7].

We argue that human knowledge should be incorporated into test generation to make the process more effective, and models that explicitly encode the knowledge provide a good means of such incorporation. In this work, we propose the Adamant approach that conveys, through an input model, valuable knowledge of the app at hand to the test generation process cost-effectively. Guided by such knowledge, Adamant can then generate test scripts that exercise more code and detect more bugs of the app.

The input model is encoded in an extended version of the Interaction Flow Modeling Language (IFML) [8], a graphical modeling language originally designed for "expressing the content, user interaction and control behaviour of the front-end of software applications"[1]. Graphical modeling languages, with intuitive notations and rigorous semantics, have been proven successful in modeling traditional desktop applications, but the same success has not been witnessed on the Android platform. Compared with desktop applications, Android apps' executions highly rely on the graphical user interfaces (GUIs) of apps, hence it is more straightforward for engineers to treat GUI elements as first-class citizens, while events as associated to GUI elements and therefore auxiliary, in modeling apps. However, existing modeling mechanisms like event-flow graphs [4] and finite-state machines [9] focus more on events or actions firing the events, rather than GUI elements.

As a newly standardized graphical language for modeling user interactions, IFML provides already mechanisms to model most aspects of mobile app GUIs, however it also suffers from a few limitations that make modeling Android apps less straightforward and IFML models less useful for Android test generation. For example, the modeling of Android-specific GUI elements like NotificationAreas and events like SwipeEvent and PinchEvent is not readily supported by IFML. More importantly, the language does not support the modeling of updates to GUI-related application states. Adamant extends IFML accordingly to address the limitations.

Given the model for an Android app in extended IFML (E-IFML), Adamant traverses the model to produce event sequences for the app, with the feasibility of each event sequence constrained by a group of conditions on the inputs to the

[1] `http://www.ifml.org/`

model. Adamant then employs a constraint solver to find appropriate values for the inputs so that the conditions are satisfied, and translates each event sequence with the corresponding input values into a test script.

We implemented the approach into a tool, also called Adamant, that offers a graphical front-end for E-IFML model construction and a back-end for Android test generation and execution. To evaluate the performance of Adamant, we applied it to generate test scripts for 10 open source Android apps. Adamant generated over 130 test actions per minute on average, and the produced test scripts managed to cover around 68% of the code and reveal 8 real bugs. We also applied other state-of-the-art test generation tools like Monkey [6], AndroidRipper [10], and Gator [11] to the same apps. Experimental results show that Adamant significantly outperformed all the three tools in terms of both statement coverage achieved and number of bugs detected. In another small-scale controlled experiment, we compared test generation using Adamant and manually. As the result, the two approaches achieved comparable cost-effectiveness.

While Adamant expects as the input E-IFML models for the apps under testing and the construction of those models takes additional time, the benefits of adopting a model-driven testing approach like Adamant are multifold and beyond just test generation. Precise modeling forces developers to devise an explicit design for an app, which is one of the key ingredients for successful software development [12]. Besides, models can also improve the development process, e.g., by fostering the separation of concerns [13], improving the communication between participants in a project [14], enabling the analysis, verification, and validation of the apps at design time [15, 16], and accelerating the development of apps through code generation [16]. Such benefits also add extra value to the Adamant approach.

The contributions of this paper can be summarized as follows:

- *Theory*: To the best of our knowledge, E-IFML is the first extension of IFML that enables the generation of concrete test scripts for Android apps;
- *Tool*: We implemented the Adamant technique into a tool, also named Adamant, that automatically generates test scripts for Android apps based on models in E-IFML. The tool is publicly available at:
  `https://github.com/ADAMANT2018/ADAMANT`.
- *Experiments*: We empirically evaluated Adamant on 10 open source Android apps; The generated test scripts achieved high code coverage on object apps and detected real bugs.

The remainder of this paper is organized as follows. Section 2 uses an example to introduce the core concepts in IFML. Section 3 introduces the extensions to IFML for facilitating Android GUI modeling. Section 4 formally defines E-IFML models. Section 5 presents the detailed process of Android test generation based on E-IFML models. Section 6 evaluates Adamant with real-world apps. Section 7 reviews related work and Section 8 concludes the paper.

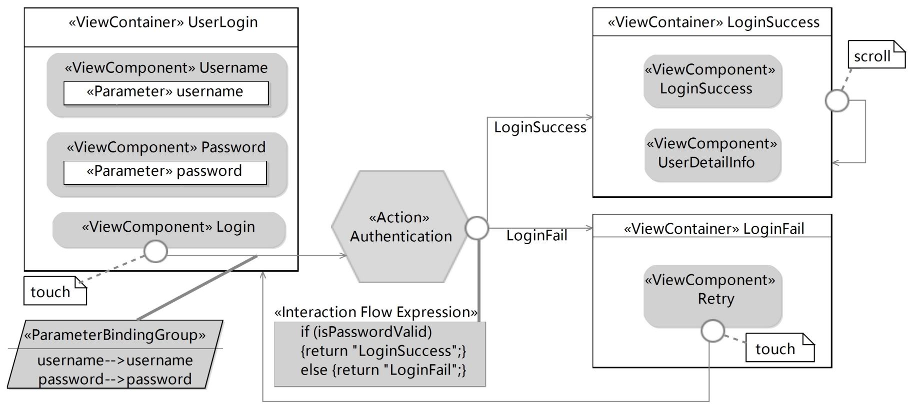

Figure 1: An IFML model specifying a user login procedure.

2. The Interaction Flow Modeling Language

The Interaction Flow Modeling Language (IFML) supports the modeling of user interfaces for applications on various types of platforms by defining both a set of generic core concepts that are common to those user interfaces and extension mechanisms to allow the refinement of the semantics of those concepts. This section briefly introduces IFML concepts that are essential in modeling Android app GUIs. An IFML model specifying the user login procedure through a GUI is presented in Figure 1 as a running example.

In IFML, *ViewContainer*s are used to help organize elements on GUIs. A ViewContainer may comprise other ViewContainers or *ViewComponent*s that display contents and support user interactions.

In the example, a ViewContainer UserLogin contains three ViewComponents, among which Username and Password are used for accepting textual inputs from the user. To facilitate the reference to those inputs in other parts of the model, two typed *Parameter*s username and password are associated to the two components. ViewContainers and ViewComponents are collectively referred to as *view elements* in this work.

*Event*s, denoted using small circles, can be triggered on view elements and handled by *Action*s, denoted using hexagons. An action represents a possibly parameterized piece of business logic. Actions are connected with their corresponding events through *InteractionFlow*s, denoted using directed lines pointing from the latter to the former, and their parameters are associated with the related InteractionFlows. In IFML, input-output dependencies between view elements or between view elements and actions are represented by *ParameterBindings*; A *ParameterBindingGroup* is simply a group of ParameterBindings. For a simple event only causing a view transition, an InteractionFlow can also be used to directly connect the event and the destination view. In the example, when a touch event is triggered on ViewComponent Login, action Authentication will be executed to handle the event. The ParameterBindingGroup associated with the corresponding InteractionFlow binds parameters username and password of UserLogin to those with the same names in the action[2]. Action Authentication then decides whether the credentials are valid or not and triggers an *ActionEvent*, i.e., a specific type of event, upon its completion. There are two InteractionFlows associated with the ActionEvent, which one to follow is decided by the evaluation result of the ActionEvent's *Interaction Flow Expression*, denoted using a rectangle. Following one of the two InteractionFlows, either ViewContainer LoginFail or ViewContainer LoginSuccess will be shown. On ViewContainer LoginFail, a touch event triggered on ViewComponent Retry will transit the app back to UserLogin so that the user can try to login again; On ViewContainer LoginSuccess, a scroll event will cause the app to refresh the display of ViewComponent UserDetailInfo in the container.

The example demonstrates the usage of core concepts in IFML. Given the direct correspondence between those concepts and common notions in GUI design, GUI modeling in IFML is natural and straightforward in many cases. IFML, however, lacks a good support for modeling certain Android view elements, events, and actions, which adds to the difficulties in modeling Android apps with IFML and makes the resultant models less useful for test generation. In the next section, we extend the language so that it can be readily used in Android app modeling and test generation.

3. IFML Extension for Android App Modeling

To better model the interactions between users and Android apps, we extend existing mechanisms provided by IFML from three aspects regarding view elements, events, and user interactions.

3.1. Extensions to View Elements

In order to improve IFML's expressiveness in modeling Android specific contents, we extend the concepts of ViewContainer and ViewComponent as illustrated in Figure 2 and Figure 3, respectively. In particular, we add two subclasses of ViewContainer called *AndroidAppContainer* and *AndroidSystemContainer*. An AndroidAppContainer defines an area on an Android GUI that corresponds to a *Screen*, a *ToolBar*, a *Web*, or a navigation *Drawer*[3]; An AndroidSystemContainer defines an area called *NotificationArea*, which is managed by the system, instead of by individual apps, and displays notifications from the Android system. These specific containers are introduced to help restrict the components that can appear in certain GUI areas and facilitate locating widgets during testing. For example, large images or long texts should not be used in a ToolBar, and system notifications should be shown in the NotificationArea.

[2]Parameters are all unique, even though they apparently share the same name.

[3]A navigation drawer is a sliding panel that can be used to show the app's navigation menu. It is hidden when not in use.

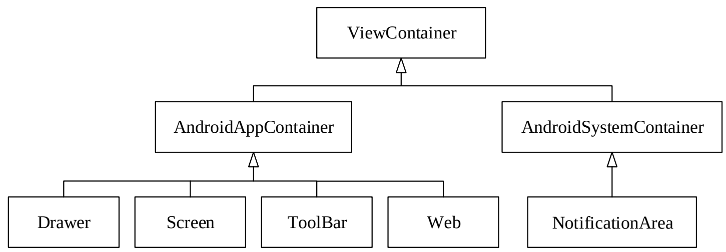

Figure 2: The extension to ViewContainer.

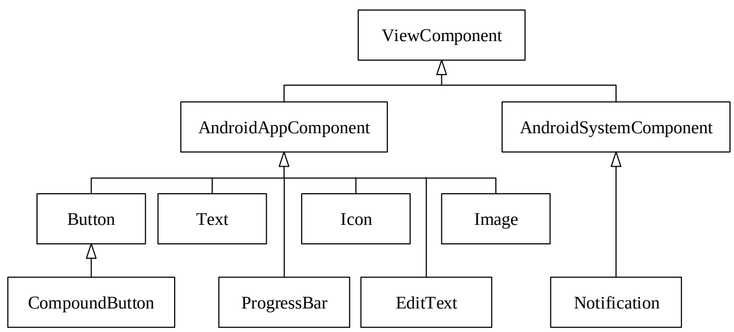

Figure 3: The extension to ViewComponent.

The concept of ViewComponent is extended in a similar way. As shown in Fig. 3, we add *AndroidAppComponent* and *AndroidSystemComponent* for components specific to Android apps and the Android system, respectively. AndroidAppComponents include common components on Android GUIs, such as *Button*s, *Text*s, *Icon*s, *Image*s, *ProgressBar*s, *EditText*s, and *CompoundButton*s. Since a NotificationArea is dedicated for displaying notifications from the Android system, we introduce *Notification* as a type of AndroidSystemComponent to model the notification content list in notification areas.

3.2. Extensions to Events

The Android platform supports a rich set of gestures that can be performed on view elements, each of which triggers its own type of event to be handled separately. Thus, it is necessary to distinguish the different types of events when modeling such apps. Events resulting from user interactions are modeled using ViewElementEvents in IFML, our extension to which is shown in Fig. 4.

A subclass *AndroidElementEvent* is introduced to model the following types of events: *TouchEvent*, *DoubleTapEvent*, *LongPressEvent*, *PinchEvent*, *ScrollEvent*, *SwipeEvent*, and *DragDropEvent*, each event type for modeling a particular type of user gesture. Besides, *InputEvent* is added to model text input events. We associated attributes with some event types to accommodate extra information about the gestures. For instance, each LongPressEvent has

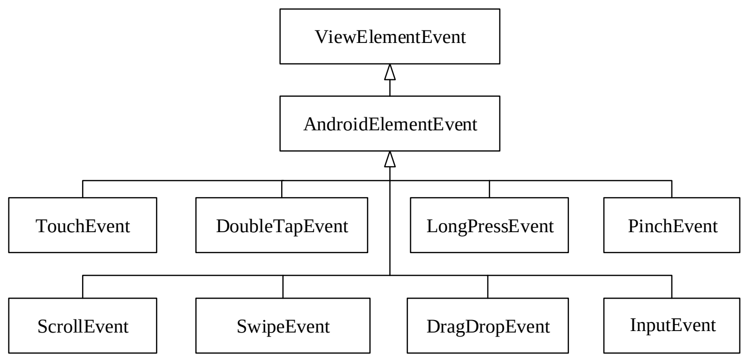

Figure 4: The extension to ViewElementEvent.

an attribute called *length* to specify the duration of the long press gesture, each SwipeEvent has an attribute called *direction* to specify how the swipe gesture was made, and each ScrollEvent has two attributes called *startingPoint* and *endingPoint* to specify where the scroll gesture starts and ends on the screen. These extended events enable us to model gestures on Android apps easily.

To fully capture the behaviours of an Android app, system events must also be considered in the app's model, since those events may occur at different points in time and affect the app's behaviour in various ways. For instance, an event caused by the sudden disconnection of Wifi may interrupt the normal use of an Android app whose functionality hinges on good network connection, and an event caused by a change of the device orientation will result in an adjustment to the current view on the screen. As shown in Fig. 5, we extend SystemEvents with a subclass *AndroidSystemEvent*, which has 5 subclasses itself. A *SensorEvent* occurs when the sensed condition is changed. It can either be a *MotionSensorEvent*, an *EnvironmentSensorEvent*, or a *PositionSensorEvent*, each of which can be further divided into more specific classes. A *ConnectionEvent* happens when a connection is established or broken, and it can be a *BlueToothEvent*, a *NFCEvent*, a *WifiEvent*, a *P2Pevent*, or a *USBEvent*. The meaning of the rest events, including *BatteryEvent*, *NotificationEvent*, and *StorageEvent*, are straightforward. The support for system events is essential for modeling Android apps. Most state-of-the-art modeling approaches only focus on events that are internal to apps while ignore system events, but system events can also affect apps' behaviours and therefore should not be ignored in modeling.

3.3. Extensions to Expression and Action

IFML uses Expressions, Actions, and external models to express the internal logic of applications [8]. To figure out the details of an app's internal logic, one needs to refer to the corresponding domain models, e.g., in the form of UML diagrams. Such design makes the modeling process more error-prone and the models harder to comprehend, since the internal logic is usually scattered throughout the whole application. To address, at least partially, that problem

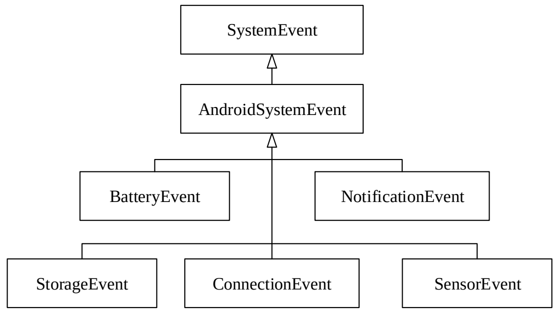

Figure 5: The extension to system event.

while without sacrificing the user-friendliness of E-IFML, we allow Java expressions to be directly used in E-IFML models to encode simple calculation. The choice of the Java programming language is motivated by the fact that most Android apps are written in Java. In this way, expressions devised in the design phase can be easily reused to implement the logic later, and expressions from the implementation can be reused when constructing models, e.g., through reverse engineering. The extension is sufficient for modeling simple logics behind many GUI interactions and significantly increases the expressing power of E-IFML. Such extension also enables test generation to take those logics into account and produce scripts that exercise different behaviors of apps, as we describe in Section 5.

Similarly, an Action in IFML represents a piece of business logic triggered by an Event, while the detailed logic is typically described in other behavioural models [8] and stored in its attribute *dynamicBehaviour*. Accordingly, we add an *executionExpression* attribute to each Action object. An execution expression models the influences of an Action on the app GUI, using a group of Java expressions with side-effects (to parameters). Correspondingly, a subclass of Expression called *ExecutionExpression* is added.

3.4. An Example E-IFML Model

The above extensions enable us to easily model concrete user interactions on Android apps using E-IFML. For example, Figure 6 shows the E-IFML model for the user login procedure of an Android app, as described in Figure 1.

In Figure 6, the original ViewContainers and ViewComponents are now modeled as Screens, EditTexts, Texts, and Buttons; The three events on view elements Login, LoginSuccess and Retry are specified as TouchEvent, ScrollEvent, and TouchEvent, respectively; The internal business logic of Action Authentication is now represented as an ExecutionExpression, which defines how a method check is invoked on username and password to decide the validity of the credentials.

Note that the (possibly complex) internal logic of check is encapsulated into a Java method and invoked in the example, which showcases the expressiveness of E-IFML expressions. Complex expressions or expressions invoking complex computations, however, will impose challenges to the constraint solving process

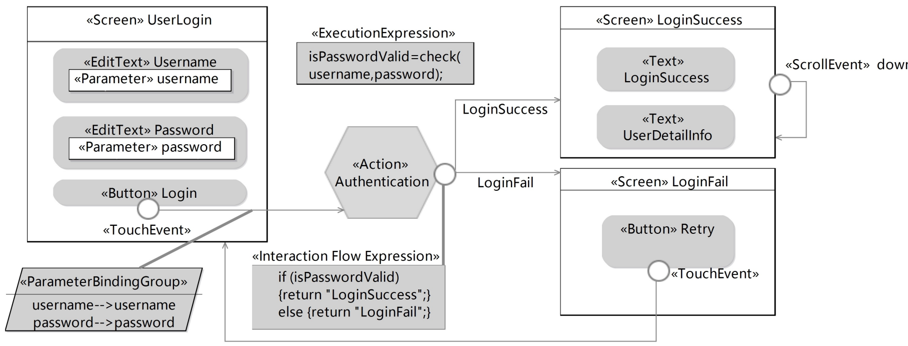

Figure 6: An E-IFML model specifying the user login procedure using an Android app.

(Section 5) if the E-IFML model is to be used for generating tests, so it is important to build the models at proper abstraction levels in practice to facilitate both model construction and model-driven testing.

The E-IFML model is not only more informative than the corresponding IFML one, but also more instructive for model-based automated test generation. For instance, to locate view element LoginSuccess of type *Text* on Screen LoginSuccess, a test generator will only need to check TextViews, but not widgets of other types, on the GUI, and the type of an event helps to regulate the type of test action that should be generated to trigger the event.

4. Formal Definition of Extended IFML Models

This section presents the formal definition and semantics of extended IFML (E-IFML) models, based on which Section 5 develops techniques that use E-IFML models to guide Android app test generation.

We use $\mathcal{T}_W$ to denote the set of ViewComponent types (including, e.g., $\mathcal{T}_{\text{Text}}$, $\mathcal{T}_{\text{Button}}$, and $\mathcal{T}_{\text{List}}$), $\mathcal{T}_C$ to denote the set of ViewContainer types (including, e.g., $\mathcal{T}_{\text{Drawer}}$, $\mathcal{T}_{\text{Screen}}$, and $\mathcal{T}_{\text{Toolbar}}$), T_E to denote the set of AndroidElementEvent types (including, e.g., $T_{\text{TouchEvent}}$, $T_{\text{ScrollEvent}}$, and $T_{\text{LongPressEvent}}$), and T_S to denote the set of AndroidSystemEvent types (including, e.g., $T_{\text{BatteryEvent}}$, $T_{\text{StorageEvent}}$, and $T_{\text{SensorEvent}}$). $T = T_E \cup T_S$ and $\mathcal{T} = \mathcal{T}_W \cup \mathcal{T}_C$ are then the sets of all event types and view types supported in E-IFML, respectively.

4.1. The Model

An E-IFML model is a 7-tuple $\langle P, E, W, CV, A, \mathcal{E}, F \rangle$, with its components formally defined as the following.

P is the set of unique parameters and $E = E_A \cup E_I \cup E_E$ is the set of expressions in the model, where 1) E_A is the set of ActivationExpressions, 2) E_I is the set of InteractionFlowExpressions, and 3) E_E is the set of ExecutionExpressions.

W is the set of all atomic views in the model. An *atomic view* (i.e., a ViewComponent) w is a 4-tuple $\langle p, e_a, t_w, c\rangle$, where 1) $p \in P$ is the parameter associated with w; 2) $e_a \in E_A$ is an ActivationExpression for w. That is, w is only enabled if e evaluates to **true**; 3) $t_w \in \mathcal{T}_W$ is the type of w; 4) c is the *composite view* that immediately contains w.

CV is the set of all composite views in the model. A *composite view* (i.e., a ViewContainer) c is a 4-tuple $\langle W_c, P_c, t_c, c_c\rangle$, where 1) $W_c \subseteq W$ is the set of atomic views within c; 2) $P_c \subseteq P$ is the set of parameters associated with c; 3) $t_c \in \mathcal{T}_C$ is the type of c; 4) c_c is the composite view that immediately contains c. A composite view c contains another composite view c', denoted as $c' \lll c$, if and only if $c'.w_{c'} \subseteq c.w_c$; c *immediately* contains c', denoted as $c' \prec c$, if $c' \lll c$ and no composite view c'' ($c'' \notin \{c, c'\}$) exists such that $c' \lll c'' \wedge c'' \lll c$. The contains and immediately-contains relation can be easily extended to work also between composite views and atomic views.

Consider the example in Figure 6. Let c_u be the composite view for ViewContainer UserLogin, w_u and w_p be the atomic views for ViewComponents username and password, $c_u.p_u$ and $c_u.p_p$ be the parameters associated with w_u and w_p[4]. We have $w_u = \langle c_u.p_u, true, \mathcal{T}_{\text{EditText}}, c_u\rangle$ and $c_u = \langle\{w_u, w_p\}, \{c_u.p_u, c_u.p_p\}, \mathcal{T}_{\text{Screen}}, \text{NULL}\rangle$.

A is the set of actions in the model. An *action* a is a pair $\langle P_a, E_a\rangle$, where 1) $P_a \subseteq P$ is the set of parameters associated with a; 2) $E_a \subseteq E_E$ is the set of expressions that will be evaluated when a is executed.

Composite views and actions are collectively referred to as *event contexts*, since events can be triggered on both composite views and actions. That is, the set EC of event contexts is equal to $CV \cup A$. Given an expression e defined in an event context $ec \in EC$, we denote the evaluation result of e w.r.t. ec as $[\![e]\!]_{ec}$.

$\mathcal{E}$ is the set of all events in the model. An *event* ϵ is a 6-tuple $\langle ec_\epsilon, t_\epsilon, d_\epsilon, ae_\epsilon, e_\epsilon, F_\epsilon\rangle$, where 1) $ec_\epsilon \in EC$ is the event context on which ϵ is triggered; 2) $t_\epsilon \in T$ is the type of ϵ; 3) d_ϵ is the data associated with ϵ, whose meaning is determined by t_ϵ; For example, information like $durationTime$ will be stored in d_ϵ if $t_\epsilon = T_{\text{LongPressEvent}}$. 4) $ae_\epsilon \in E_A$ is the ActivationExpression associated with ϵ; Similar to the case for view components, ϵ is only enabled if ae_ϵ evaluates to **true**; 5) $e_\epsilon \in E_I$ is the InteractionFlowExpression of ϵ, if any; 6) F_ϵ is the set of interaction flows starting from ϵ. The flow to be executed is determined based on the evaluation result of e_ϵ and available flows in F_ϵ;

F is the set of all interaction flows in the model. An *interaction flow* f is a 4-tuple $\langle \epsilon_f, c_f, ec_f, B_f\rangle$, where 1) $\epsilon_f \in \mathcal{E}$ is the event initiating f; 2) c_f is a constant value; f is only executed if the interaction flow expression of its initiating event $\epsilon_f.e$ evaluates to c_f. 3) $ec_f \in EC$ is the destination context of f; We refer to the triggering context of f's initiating event, i.e., $\epsilon_f.ec$, as the *source context* of f. 4) $B_f \subseteq P \times P$ is the group of parameter bindings for f.

Continue with the example in Figure 6. Action Authentication can be denoted as $\alpha = \langle\{\alpha.p_u, \alpha.p_p\}, \varnothing\rangle$. The event triggering the action is $\epsilon_\alpha = \langle c_u, T_{\text{TouchEvent}},$

[4]We refer to a parameter p defined in context c as $c.p$.

$null, true, null, \{f_\alpha\}\rangle$, where $f_\alpha = \langle \epsilon_\alpha, true, \alpha, \{c_u.p_u \to \alpha.p_u, c_u.p_p \to \alpha.p_p\}\rangle$ is the interaction flow connecting ϵ_α and α. Here we use $\to$ to denote the binding relation between parameters.

Based on these definitions, the behaviors of an app, driven by events triggered by users or generated by the system, can then be modeled as paths of a finite automaton $\mathcal{M} = \{\Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{T}, \boldsymbol{F}\}$, where

- $\Sigma = \mathcal{E} \times F$ is the set of event and interaction flow pairs in the app;
- $\boldsymbol{S} = EC = CV \cup A$ is the set of event contexts in the app;
- $s_0 \in CV$ is the initial composite view of the app;
- $\boldsymbol{T} \subseteq \boldsymbol{S} \times \Sigma \times \boldsymbol{S}$ is the set of transitions between event contexts;
- $\boldsymbol{F} = CV$ is the set of composite views where the handling of user interactions terminates.

Note that we regard all composite views, but no action, as acceptable final states of the automaton, since a user may decide to exit an app at any view of the app, while an app should always be able to arrive at a composite view after finishing the execution of an action.

4.2. Well-formedness and Feasibility of Paths

Given a transition $\tau = \langle ec_b, \langle \epsilon, f\rangle, ec_e\rangle \in \boldsymbol{T}$, τ is *well-formed* if the following conditions are satisfied:

C1. Event ϵ can be triggered on ec_b. More specifically, if $ec_b \in CV$, then the event context on which ϵ is triggered is within ec_b, i.e., $\epsilon.ec \ll ec_b$; If $ec_b \in A$, then $\epsilon.t$ should be of type *ActionEvent*;

C2. f starts from ϵ, i.e., $f \in \epsilon.F$; and

C3. The destination event context of f is ec_e, i.e., $f.ec_f = ec_e$.

Due to the constraints imposed, e.g., by activation expressions on atomic views, a well-formed transition τ may not be actually *feasible* during app executions. Particularly, τ is feasible if and only if there exists a function $\alpha : P \to V$ that assigns a concrete value $\alpha(p)$ to each input parameter $p \in P$ such that the following conditions are satisfied:

C4. Event ϵ is enabled in context ec_b, i.e., $[\![\epsilon.ae]\!]_{ec_b} = \textbf{true}$;

C5. the interaction flow expression $\epsilon.e$ evaluates to $f.c$ in context ec_b and therefore f is activated, i.e., $[\![\epsilon.e]\!]_{ec_b} = f.c$.

Correspondingly, a sequence $\rho = \rho_1, \rho_2, \ldots, \rho_n$ of transitions ($\rho_i \in \boldsymbol{T}$, $1 \leqslant i \leqslant n$) constitutes a *well-formed* path on $\mathcal{M}$ if and only if 1) $\rho_1.src = s_0$, 2) $\rho_i.dest = \rho_{i+1}.src$ ($1 \leqslant i < n$), and 3) each ρ_j ($1 \leqslant j \leqslant n$) is *well-formed*; a

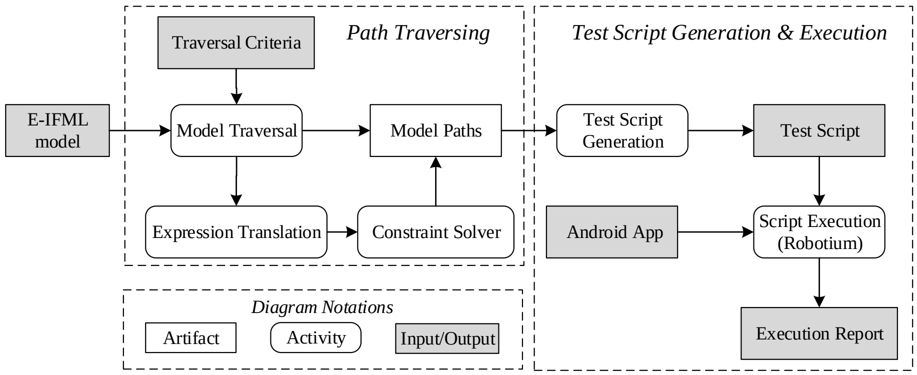

Figure 7: The overview of ADAMANT.

well-formed path ρ is *feasible* if and only if there exists a function $\alpha : P \to V$ that renders all transitions on ρ feasible.

To decide whether a well-formed path ρ is feasible or not, we collect the group G_ρ of constraints along ρ on $\mathcal{M}$'s parameters, and find solutions to the constraints using an off-the-shelf solver. Execution expressions and parameter binding are processed in a similar way during this feasibility analysis as in symbolic execution [17]. The constraints can be used not only to determine the feasibility of a path, but also to find out actual assignments to the input parameters that will realize the path, if it is decided to be feasible.

5. Test Generation Based on E-IFML Models

Based on the above definitions, we propose the Adamant approach for automated Android app testing based on E-IFML models. A high-level overview of Adamant is depicted in Figure 7. Taking the E-IFML model of an Android app as the input, Adamant constructs the corresponding finite state automaton $\mathcal{M}$, traverses the automaton in a depth-first manner to generate paths of $\mathcal{M}$ with constraints G for their feasibility. Adamant then employs an off-the-shelf constraint solver to find solutions to G, i.e., assignments to parameters in $\mathcal{M}$. If successful, Adamant then combines the paths and corresponding assignments to produce concrete test scripts, which are then passed to script automators such as Robotium [18] to be executed on Android apps.

5.1. Path Generation

Algorithm 1 outlines the main logic of path construction. Recursive function traverse takes four arguments: the current *context* of traversal, (symbolic or concrete) *values* of the parameters in the model, the *path* under construction, and all the feasible *paths* constructed.

During the execution of traverse, if *path* should be extended (Line 2), we get the set of candidate events from *context* (Line 3) and, for each event and

Algorithm 1: Algorithm for path generation.

```
1  Function TRAVERSE(context, values, path, paths):
2      if SHOULDEXTEND(path) then
3          foreach event ∈ GETEVENTS(context) do
4              foreach f ∈ event.F do
5                  values′ ← EVAL(context, values, event, f);
6                  path′ ← path · ⟨context, event, f.ec⟩;
7                  TRAVERSE(f.ec, values′, path′, Paths);
8              end
9          end
10     else
11         if HASNEWEVENT(path, paths) then
12             constraints ← path.getConstraints();
13             solution ← SOLVE(constraints);
14             if solution ≠ null then
15                 paths.ADD(path, solution);
16             end
17         end
18     end
19     return paths
20 end
```

its associated interaction flow (Lines 3 and 4), ADAMANT symbolically triggers the event and follows the interaction flow (Line 5), causing updates to *values* (Line 5) and *path* to be extended to $path'$ (Line 6). Afterwards, the recursion continues using the updated *context*, *values*, and *path* (Line 7). Once the algorithm decides the current path no longer needs to be extended, *path* is checked against all paths from *Paths* to find out if it exercises any new event (Line 11). When yes, the constraints collected along *path* is send to a solver (Lines 12 and 13). If a solution can be found for the constraints, *path* is feasible and gets added to *Paths* (Line 15).

To get all the feasible concrete paths on $\mathcal{M}$, we call function TRAVERSE with arguments $context = s_0$, $values = \varnothing$, $path = [\,]$ (i.e., an empty sequence), and $paths = \{\}$ (i.e., an empty map). The return value contains paths that start from the initial composite view s_0 and satisfy the requirements regarding path constitution, as described below.

To strike a balance between cost and effectiveness, we adopt event coverage as our test adequacy criterion in path generation. A path is considered useful only if it can help increase the overall event coverage, i.e., if it contains events that were not covered by other paths. For that, we use two parameters *Maximum Events* and *Maximum Duplicate Events* to restrict the length of each path:

- *Maximum Events* specifies the maximum number of events that are fired in a path. This parameter directly restricts the length of each path.

- *Maximum Duplicate Events* specifies the maximum number of duplicate events that are fired in a path.

5.2. *Test Script Generation*

When generating test cases from paths of $\mathcal{M}$, only events are needed, while event contexts can be ignored, since they correspond to the expected results of handling the events. Events in a path can be transformed in order into actions of test scripts, which can be executed later by testing tools such as ROBOTIUM.

```xml
<path id="1">
  <event order="1">
    <component type="EditText" index=0/>
    <operation type="Input" reference="2018ADAMANT"/>
  </event>
  <event order="2">
    <component type="EditText" index=1/>
    <operation type="Input" reference="qwerty"/>
  </event>
  <event order="3">
    <component type="Button" text="Login"/>
    <operation type="Touch"/>
  </event>
  <event order="4">
    <component type="Screen"/>
    <operation type="Scroll" reference="Down"/>
  </event>
</path>
```

Listing 1: Sample sequence of events.

```java
public class Testcases
      extends ActivityInstrumentationTestCase2{
   private Solo solo = null;
   private static Class<?> launcherActivityClass;
   ...
   public void testcase001() throws Exception {
      solo.typeText(0, "2018ADAMANT);
      solo.typeText(1, "qwerty123456");
      solo.clickOnText("Login");
      solo.scrollToSide(Solo.Down);
      ...
   }
}
```

Listing 2: ROBOTIUM test script.

Listing 1 shows a sample sequence of events generated by the technique described in Section 5.1. Every *event* in a *path* has two sub-elements: *component* and *operation*. A component pinpoints a target UI widget or UI component by specifying the *type*, *id* and *text*, while an operation provides the *type* and parameter information about the action to be applied on the component. In the example, the first two events (Lines 2–9) are triggered by inputting "2018ADAMANT" and "qwerty123456" in the EditTexts of index 0 and 1, the third event(Lines 10-13) is triggered by touching on the Button with text "Login" to submit the inputs, the last event (Lines 14–17) is triggered by scrolling down the screen. Here string literals like "2018ADAMANT" and "qwerty123456" are provided as optional values for the EditTexts in the model. Besides of looking for valid inputs through constraint solving, ADAMANT also utilizes such information, if available, when generating paths.

ADAMANT translates the sequence of events in Listing 1 into the test script shown in Listing 2. In particular, Lines 7-10 in method `testcase001` correspond to the four events from Listing 1, respectively. Test scripts like these can be fed to test automation frameworks like ROBOTIUM to drive the execution of the target app.

If the execution of test script fails, ROBOTIUM records information about the exception causing the failure, which can be used together with Android's logcat mechanism to facilitate the debugging process. ADAMANT also records screenshots of the app during testing, which can be used by test engineers, e.g., to manually confirm if a script tests interesting behaviours.

5.3. Tool implementation

We implemented the approach into a tool also called ADAMANT, based on the Eclipse Modeling Framework(EMF) and the Sirius API [19]. In this way, the Eclipse diagram editor can be leveraged as the graphical E-IFML model editor, and the conversion from user-built models to XML files can be easily supported using Sirius. The backend of ADAMANT takes the XML files as the input and generates test scripts.

As Section 3.2 shows, ADAMANT supports the modeling of most Android events. However, whether it can produce the corresponding gestures to these different types of events depends on the capability of the script executor. In the current implementation, ADAMANT uses ROBOTIUM as the script executor, since ROBOTIUM supports most types of events, such as click, scroll and rotation. For certain types of events with attributes, besides of using ADAMANT's default attribute values, users can also provide their own values to customize the corresponding gestures. For example, the scroll gesture corresponding to a ScrollEvent is augmented with two attributes $startingPoint$ and $endingPoint$ encoding the positions where the gesture starts and ends, respectively. The handling of time duration between consecutive events is delegated to ROBOTIUM. ROBOTIUM has a "wait or time-out" mechanism to execute events. After triggering an event, it would wait for a given time until the component on which the next event should be activated is visible. An error occurs if the time is up before the component becomes visible. ADAMANT also allows a user to set a customized waiting time before/after executing an event. As the experimental evaluation in Section 6 shows, using default settings of ADAMANT can achieve good performance.

Adamant employs the Z3 constraint solver [20] to find solutions to path constraints. Since constraint solving can be highly time-consuming, Adamant caches the constraints and their solutions for better performance. The basic idea is that it identifies sub-groups of constraints that are independent from the others. If a sub-group of constraints were never solved before, Z3 is invoked to find solutions for the constraints, and the results (including whether there exists a solution and, when yes, what the solutions are) are stored together with the constraints into a map. If a sub-group of constraints was already solved before, the results are directly retrieved from the map. Since many events in E-IFML models depend on a fixed number of parameters, executions that differ only in other parameters then share the independent sub-groups of constraints related to those events. As the result, the number of constraint combinations associated with different paths is mush smaller than that of all possible combinations, and reusing constraint solving solutions significantly improves the overall performance of Adamant, as demonstrated by the experimental evaluation of the tool in Section 6.

6. Evaluation

The experimental evaluation of Adamant assesses to what extent Adamant facilitates test generation for Android apps, and it aims to address the following research questions:

- **RQ1:** How effective is Adamant?
- **RQ2:** How efficient is Adamant?

  In RQ1 and RQ2, we evaluate the effectiveness and efficiency of Adamant in Android test generation from a user's perspective.
- **RQ3:** How does Adamant compare with other Android test generation tools?

  In RQ3, we compare Adamant with three state-of-the-art tools for Android test generation: Monkey, AndroidRipper, and Gator. Monkey randomly generates event sequences for Android apps and, although simple, it outperforms most other existing tools that are publicly available in the area [5]; AndroidRipper implements an opposite strategy by extending a tool that automatically explores an app's GUI to generate tests that exercise the app in a structured manner [10]. Unlike Monkey or AndroidRipper that build test scripts via dynamic analysis, Gator [11] employs static analysis to model GUI related objects and events of Android apps, and it has been applied to Android test generation [21].
- **RQ4:** How does constraint and solution caching impact Adamant's efficiency?

Constraint solving is time-consuming and can greatly degrade ADAMANT's efficiency if not used sparingly. In view of that, ADAMANT employs constraint and solution caching when generating execution paths from E-IFML models (Section 5.3). In RQ4, we zoom in on this design choice of ADAMANT and study whether and to what extent constraint and solution caching helps improve ADAMANT's efficiency.

6.1. Experimental Objects

To collect the apps to be used in the experiments, we first summarized apps from two lists of open source Android apps, one on Wikipedia [22] and the other on GitHub [23], into 10 categories: Browser, Communication, Security, Multimedia, Reading, Education, Tool, Weather, Productivity and Other. Then, we randomly select one app from each category that 1) has been released in Google Play, FDroid or GitHub, and 2) has at least 100 stars in GitHub, and use the latest versions of the selected apps (as of September 2018) as our objects. Such selection process is to ensure the diversity of objects. Table 1 lists, for each object, the name (App), the version (Ver), the category (Category), the size in lines of code (LOC), and the number of Activities (#Act). The app size ranges from about just one thousand to over 70 thousand lines of code, which illustrates from another aspect the diversity of experimental objects.

6.2. Experimental Subjects

We recruit ten third-year undergraduate students majoring in Software Engineering to build E-IFML models for the selected object apps. All these students gained basic knowledge in mobile development and UML from their previous studies, but received no exposure to IFML before participating in the experiments. Such selection of subjects is acceptable, since previous study found out that students and professionals perform similarly on approaches that are new to them [24], while the task of constructing E-IFML models is new for both these students and professional developers.

Table 1: Apps used as objects in the experiments.

App	Ver	Category	LOC	#Act
Bookdash	2.6.0	Education	7501	6
Connectbot	1.9.2-80	Communication	28791	12
Goodweather	4.4	Weather	5262	7
I2P	0.9.32	Security	24106	11
Kiwix	2.2	Reading	11270	8
Lightning	4.4.0.24	Browser	21017	8
Omninotes	5.4.1	Productivity	20305	9
Owncloud	2.0.1	Other	72862	12
Ringdroid	2.7.4	Multimedia	5295	4
Talalarmo	3.9	Tool	1224	2
Total			197633	79

6.3. Measures

In this work, we categorize test actions from a generated test script into four groups: successful, bug revealing, inexecutable, and unreachable: A test action is considered *successful*, if it can be executed during testing, its execution successfully triggers an event, and the handling of the event is completed with no problem; A test action is *bug revealing* if, while it can be executed during testing and its execution can successfully trigger an event, the handling of the event either terminates prematurely or hangs; A test action is *inexecutable*, if it will be attempted during testing but fails to trigger any event, e.g., because the GUI element that action operates on cannot be located in the context activity or its intended event is not supported on the target GUI element; A test action is *unreachable*, if it is located after a bug revealing or inexecutable action in a test script, and therefore it is never attempted during testing. We refer to test actions that are either successful or bug revealing as *executable* actions, and those that are either bug revealing or inexecutable as *unsuccessful* actions.

Test generation techniques like Monkey and AndroidRipper incrementally construct test scripts via dynamic analysis, and each test action they generate is always executable. In contrast, techniques like Adamant and Gator first build a model for the app under testing and then utilize the model to guide test script generation. In case the model does not comply with the app, generated test actions may 1) be inexecutable, 2) reveal bugs in either the app or the model, and/or 3) leave actions in the same script but after them unreachable.

Let $\#\mathrm{A}$, $\#\mathrm{A_s}$, $\#\mathrm{A_b}$, $\#\mathrm{A_i}$, $\#\mathrm{A_u}$, and $\#\mathrm{A_e}$ be the number of all, successful, bug revealing, inexecutable, unreachable, and executable test actions in a test suite, respectively. We have $\#\mathrm{A_e} = \#\mathrm{A_s}+\#\mathrm{A_b}$ and $\#\mathrm{A}=\#\mathrm{A_e}+\#\mathrm{A_i}+\#\mathrm{A_u}$.

To evaluate the *effectiveness* of a test generation approach from a user's perspective, we assess the size and quality of the test suites produced by the approach in terms of four commonly used measures [5]:

- #K: the number of test scripts they contain;
- #A: the number of test actions they contain;
- #B: the number of unique bugs they reveal;
- %C: the statement coverage they achieve.

During the experiments, we record the time $\mathrm{T_g}$ in minutes that each tool takes to generate the tests. Note that, for test generation based on dynamic analysis, generated tests are executed along the way, so $\mathrm{T_g}$ includes the test execution time. For test generation based on static analysis, test execution, however, is typically not part of the generation process. We therefore record in addition the time $\mathrm{T_e}$ in minutes that is required for the tests generated by a static tool to execute. Besides of $\mathrm{T_g}$ and $\mathrm{T_e}$, we also measure the *efficiency* of test generation using the following metrics:

- APM: the number of test actions generated per minute, i.e., $\#\mathrm{A}/\mathrm{T_g}$;
- %E: the percentage of generated test actions that are executable, i.e., $\#\mathrm{A_e}/\#\mathrm{A}$.

In the case of Adamant, since the construction of its input models requires considerable manual effort, we also measure the size of the E-IFML models used as the input for running Adamant in terms of: the number #Cn of containers, the number #Cm of components, the number #Ev of events, the number #IF of interaction flows, the number #Ex of expressions, the number #Ac of actions, and the time cost T_m in minutes for preparing them. We use #E to denote the total number of elements an E-IFML model contains, i.e., #E=#Cn+#Cm+#Ev+#IF+#Ex+#Ac.

6.4. Experimental Protocol

Before the ten undergraduate students start to build E-IFML models for the object apps, a 90-minute training session is provided to help them get familiar with E-IFML modeling. After the training is finished, each student is assigned an app randomly and asked to build an E-IFML model for the app from a user's perspective. Each model produced is then independently reviewed by two other students from the ten to ensure the correctness. The students are also required to record the time they spend in both model construction and review.

Next, Adamant is applied to the E-IFML models to generate test scripts for the ten object apps, and the generated tests are executed on the apps using Robotium. When the execution of a test action fails to start or run to its completion, Robotium will log the problem. We analyze the log and the corresponding test script to determine whether the test action is bug revealing or inexecutable: We conservatively mark the action as bug revealing only if the problematic behavior has been confirmed as an issue on GitHub.

Constraint and solution caching, as presented in Section 5.2, is enabled by default in the experiments described above, and the results are used to answer the first three research questions; To answer RQ4, we repeat the experiments with constraint and solution caching disabled. In both cases, the value of *Maximum Duplicate Events* used in path generation (defined in Section 5.1) by Adamant is empirically set to 2, while *Maximum Events* is set ranging from 5 to 11, respectively. A more thorough investigation on how these values affect the effectiveness and efficiency of Adamant is left for future work.

To make the comparison among test generation techniques more straightforward, Monkey, AndroidRipper, and Gator are applied to the same set of objects: Monkey runs on each app for at least the same amount of time as used by Adamant, and no less than ten minutes. The reason is that, Monkey was reported to hit its maximum code coverage within five to ten minutes [5]. Besides, since Monkey implements a random strategy for test generation, we repeat the experiment on each app using Monkey for 3 times and use the average of the results for comparison; AndroidRipper is configured to perform a breath-first search and allowed to run until its natural termination, with the time interval between two consecutive events being set to 1 second; Gator is also configured to run until its natural termination. Given the Window Transition Graph (WTG) produced at the end of a Gator run, we construct sequences of GUI events via a depth-first search, which are then translated to test scripts in the Robotium format. The default value for the maximum number of events

Table 2: Experimental results from applying ADAMANT on the ten apps.

App	#K	#A	%C	#B	T_g	T_e	APM	%E	E-IFML Model							T_m
									#Cn	#Cm	#Ev	#IF	#Ex	#Ac	#E	
Bookdash	19	180	84	0	0.2	7.0	900.0	100	21	32	40	41	19	5	158	240
ConnectBot	71	483	54	0	1.9	32.7	254.2	96	52	112	149	154	49	22	538	1080
Goodweather	28	174	84	1	0.5	7.9	348.0	99	28	59	60	61	10	4	222	360
I2P	41	451	70	0	0.5	39.7	902.0	100	67	97	108	113	36	14	435	660
Kiwix	35	252	78	1	0.5	16.8	504.0	100	38	121	108	115	63	33	478	600
Lightning	76	532	69	0	10.8	28.2	49.3	100	47	146	137	166	40	22	558	900
Omninotes	66	594	71	1	2.3	45.9	258.3	100	58	121	167	172	70	35	623	960
OwnCloud	82	648	57	5	9.7	34.4	66.8	98	65	136	195	203	72	53	724	1500
Ringdroid	28	210	82	0	0.7	9.3	300.0	100	15	52	57	61	19	8	212	360
Talalarmo	11	55	92	0	0.1	5.0	550.0	100	10	17	21	24	6	3	81	120
Overall	457	3579	68	8	27.2	226.9	131.6	99	401	893	1042	1110	384	199	4029	6780

each path may have is set to 3, as was done in [21]. All tests generated by each technique are considered in the comparison.

A home-brewed tool based on the Eclipse Java Development Tools (JDT) [25] is utilized to collect the statement coverage information of all the tests generated by each approach.

All experiments were conducted on a DELL laptop, running 64-bit Windows 10 Home on a 4-core, 2.6GHz, I7 CPU and 8GB RAM. Android apps were run in an emulator configured with 4GB RAM, X86_64 ABI image, and Android Lollipop (SDK 5.1.1, API level 22).

6.5. Experimental Results

This section reports on the results of the experiments and answers the research questions.

6.5.1. RQ1: Effectiveness.

Table 2 reports on the results from applying ADAMANT on the ten object apps. For each app, the table lists the measures for effectiveness as defined in Section 6.3. Overall, ADAMANT generated for the apps 11 to 82 scripts with 55 to 648 test actions, averaging to 46 scripts and 358 actions for each app.

Statement coverage. The statement coverage achieved by the generated tests varies between 54% and 92% on individual apps, amounting to 68% over all the ten apps, which suggests that ADAMANT is effective in exercising most code of the apps.

The highest coverage was achieved on app Talalarmo, which is the smallest in size among all the objects: Smaller apps tend to have fewer functionalities and are often easier to build comprehensive models for. The lowest coverage was observed on app Connectbot. While this app is not the largest in LOC, it has the most activities and a significant portion of its code is only exercised upon inputting strings in certain formats, which increases the difficulties in testing more of its code.

Bug detection. In total, 8 unique bugs in the object apps were revealed by 22 unsuccessful test actions, among which 9 caused crashes and the other

Table 3: Bugs found by ADAMANT.

ID	App	Bug	
		Sym	Description
B1	Goodweather	ISE	provider doesn't exist: network
B2	Owncloud	NPE	Method 'java.lang.String.toCharArray()' is invoked on a null object reference.
B3	Owncloud	NPE	Method 'com.owncloud.android.lib.common.operations.RemoteOperationResult.isSuccess()' is invoked on a null object reference.
B4	Owncloud	NPE	Method 'android.view.View.getImportantForAccessibility()' is invoked on a null object reference.
B5	Owncloud	INC	When exiting or going back from searching, the file list is cleared not refreshed.
B6	Owncloud	CCE	com.owncloud.android.ui.activity.FolderPickerActivity cannot be cast to com.owncloud.android.ui.activity.FileDisplayActivity.
B7	Omninotes	INC	User authentication can be bypassed by clicking on the "password forgotten" button on the login activity and then the BACK button.
B8	Kiwix	SC	Service Pico TTS crashes with error message "Fatal signal 11 (SIGSEGV), code 1, fault addr 0x7f0dda-291970 in tid 14926(com.svox.pico)".

13 were inexecutable. Specifically, 7 out of the 9 crashes happened due to bugs hidden in apps, while the other 2 were caused by the crash of ROBOTIUM when test cases tried to restart the object apps. Among the 13 failures caused by inexecutable test actions, 5 were due to unexpected conditions such as no response from remote servers caused by unreliable network, while the rest 8 happened when the target GUI elements cannot be found on the corresponding app activities, which indicates that there are discrepancies between the E-IFML models built by students and the actual app implementations. A closer look at the discrepancies reveals that 3 expressions were incorrectly specified. Recall that all the models contain in total 384 expressions (Table 2). While a systematic study on the quality of the expressions is beyond the scope of this paper and we leave it for future work, existing evidence suggests users can correctly write most expressions with reasonable effort.

Table 3 lists for each bug its ID (ID), the app it belongs to (App), its symptom (Sym), and a short description (Description). In column Sym, NPE stands for NullPointerException, CCE stands for ClassCastException, INC stands for Inconsistency, ISE stands for IllegalStateException, and SC stands for service crash.

Bugs B1, B2, B3, B4, B6, and B8 caused apps to crash during the execution of test scripts, while bugs B5 and B7 caused test execution to hang. In particular, Bug B5 was revealed in the popular file sharing app named OwnCloud by a test script that opens a directoy *dir* in the app, performs a file search, exits from the search, and then selects a file from *dir*. The test was generated, since the model of the app suggests that, when exiting from a search, the app should return to the state before the search, which is quite reasonable. However, the app failed to clear the search result and show the contents of directory *dir* when exiting from the search, making the selection of a file from *dir* infeasible and causing the test execution to hang. Bug B7 was found in a note taking app named Omninotes. To delete a locked note in Omninotes, a user needs to log in first. ADAMANT, however, was able to generate a test script that circumvents the rule by first clicking on the "password forgotten" button on the login dialog and pressing the BACK button, and then deletes the locked note without logging in. With the note deleted, a following action that operates on the note becomes inexecutable and the execution of the test script hangs. The app behaviors related to bugs B5 and B7 are both marked as buggy by the app developers on GitHub[5].

Since bugs like B5 and B7 do not cause any crashes, they will not attract any attention even if tools like MONKEY and ANDROIDRIPPER are employed. ADAMANT, however, can also help discover such bugs if they cause behaviors that contradict users' expectations.

> ADAMANT *effectively generated test scripts to exercise 68% of the object apps' statements and discover 8 unique bugs.*

6.5.2. RQ2: Efficiency.

Table 2 also lists for each object app the measures for ADAMANT's efficiency, the size information about the corresponding E-IFML model, and the time students spent to construct and review the model.

It took ADAMANT 27.2 minutes in total to generate all the test scripts for the object apps, with the average being 2.7 minutes for each app, which suggests the time cost for running ADAMANT is moderate in most cases. Only two out of the ten apps had longer test generation time than the average: Lightning and OwnCloud. Both apps are larger and more complex than the others: OwnCloud is the largest, in terms of both the number of activities and lines of code, among all the object apps, and its E-IFML model is also the most complex among the ten: the model has by far the largest number of events, information flows, and expressions, and the longest construction and review time; While Lightning is not as large or complex, it has by far the largest number of paths to examine during path generation (Section 5.1), and it takes quite some time for ADAMANT to construct and then process those paths. As expected, it takes longer to execute, than to generate, the tests. The execution time of all the generated tests amounts to 226.9 minutes, averaging to 0.5 minutes per test script and 3.8

[5] `http://www.github.com/federicoiosue/Omni-Notes/issues/372` for bug B5, and `http://www.github.com/nextcloud/android/issues/1640` for bug B7.

seconds per test action.

The APM values on most apps ranged between 250 and 1000, which suggests that Adamant is reasonably efficient in generating tests for the apps. The lowest APM values were observed on apps Lightning and OwnCloud, most likely due to the long generation time. Overall, Adamant generated 131.6 test actions per minute for the apps.

Measure %E is equal to 100% for 7 of the objects, and is above 95% for the remaining 3, indicating that most test actions generated by Adamant are indeed executable. On the one hand, such high values show that the models faithfully capture the behaviors of the apps; On the other hand, they also speak well for Adamant's capability to correctly consume the information provided by the models. We further inspected the reasons for the low percentages of the 3 apps. For apps GoodWeather and OwnCloud, bugs in their implementations rendered 1 generated test actions to be inexecutable and 15 to be unreachable. As for ConnectBot, the reason, however, lies in bugs in the constructed model: 6 test scripts generated for ConnectBot based on the faulty model failed due to the bugs, leaving 14 actions unreachable. Overall, 99% of the generated test actions are indeed executable.

To get a better understanding of the overall cost for the application of Adamant, we also examine the time spent in preparing the input models . Table 2 shows that considerable manual effort is required to construct the E-IFML models in the experiments and the modeling time is in proportion to the overall size of the resultant models: the average time needed to model a single GUI element is around 1.7 minutes (=6780/4029) across all objects, and that average time for each app varies between 1.3 and 2.1 minutes. In view of such high cost for manually constructing the models, we plan to develop techniques to (at least partially) automate the task of E-IFML model construction for Android apps in the future.

752 *On average,* Adamant *generates 131.6 test actions per minute, 99% of which are executable. The construction time of E-IFML models is in proportion to the size of resultant models, averaging to c.a. 1.7 minutes per GUI element.*

6.5.3. RQ3: Comparison with other techniques.

Table 4 presents the results of running Monkey, AndroidRipper, and Gator on the same apps. Note that the table does not report the values of all measures: 1) Since the number of generated test scripts and the length of a test script largely depend on the configurations of these tools, the table does not report measure #K; Instead, we report the more meaningful measure #A. 2) AndroidRipper does not report the total number of generated test actions to the user, so we also omit measures #A and APM for AndroidRipper in the table; 3) Measure %E is omitted for both Monkey and AndroidRipper, because test actions generated by these two tools are always executable, resulting in 100% values for the measure; 4) Neither AndroidRipper nor Gator detected any bug in the object apps, hence we also leave out column #B for the two tools in the table. Besides, AndroidRipper failed to test app Bookdash since exceptions

Table 4: Experimental results of MONKEY, ANDROIDRIPPER, and GATOR on the objects.

App	%U*	MONKEY						ANDROIDRIPPER			GATOR						
		#A	%C	%U	#B	T_g	APM	%C	%U	T_g	#A	%C	%U	T_g	T_e	APM	%E
Bookdash	22.0	9500	62	0.9	0	10.0	950	-	-	-	130	33	0.0	0.2	9.0	650	90
ConnectBot	17.0	33500	42	3.2	0	35.0	957	22	4.5	49.3	4413	18	0.0	0.6	232.9	7355	4
Goodweather	17.7	9500	62	0.8	0	10.0	950	15	0.6	5.3	102	24	0.3	0.1	8.8	1020	91
I2P	45.4	39600	27	1.2	0	41.7	950	21	0.0	21.4	609	15	0.1	1.4	38.1	435	12
Kiwix	18.1	18000	63	1.2	1	20.0	900	32	0.0	18.5	-	-	-	-	-	-	-
Lightning	9.0	28000	61	2.7	0	30.0	933	40	3.7	23.2	1191	34	0.7	12.1	82.2	98	21
Omninotes	26.6	45000	45	1.5	0	48.3	932	21	0.0	3.1	35282	22	0.4	74.4	1903.5	474	0
OwnCloud	19.1	35000	40	4.2	0	36.7	954	25	1.8	11.3	-	-	-	-	-	-	-
Ringdroid	20.5	11100	58	3.1	0	11.7	949	48	0.3	42.5	5424	51	0.9	0.1	334.6	5424	45
Talalarmo	25.5	9200	63	0.4	0	10.0	920	39	0.0	2.8	15	39	0.0	0.1	1.0	150	100
Overall	21.8	238400	45	2.8	1	253.4	941	25	1.8	177.4	47166	24	0.3	89.0	2610.1	530	7

%U*: %U achieved by ADAMANT.

were thrown when loading the app, while GATOR reported OutofMemoryError and failed to generate any test cases on apps Owncloud and Kiwix after running for 5 hours. We use dashes (-) in the table to indicate that the corresponding measures are not available, and we exclude the apps from the computation of overall measures for the two tools.

While the numbers of generated test actions vary drastically across different tools and objects (#A), the overall statement coverage achieved by the tools (%C) is in general consistent with that reported in a previous study [5]: MONKEY achieved an overall coverage of 45%, ANDROIDRIPPER 25%, and GATOR 24%. In comparison, tests generated by ADAMANT covered 68% statements of the apps, i.e., 23%, 43%, and 44% more than the other three tools.

Although MONKEY generated the most actions and at the highest speed, the statement coverage it achieved was not as high, suggesting that many such actions are redundant. We conjecture the reason is that MONKEY has no knowledge about the app's behavior and does not keep track of which behaviors were tested already. The coverage achieved by ANDROIDRIPPER and GATOR was even lower. ANDROIDRIPPER failed to recognize a considerable number of activities, leaving many GUI elements untested. The low coverage of ANDROIDRIPPER on apps like I2P, Omninotes, and Talalarmo was also because the tool crashed from time to time, causing the systematic exploration to end prematurely.As for GATOR, the low statement coverage was mainly due to the fact that only a small portion, 7% to be precise (%E), of generated test actions are executable. We inspected the failed test scripts and found the major reason for the high failing rate is that, since the extracted WTG models are often incomplete and/or incorrect, they provide little information regarding app states, e.g. whether a GUI element is visible or whether an event is feasible, to facilitate test generation. As a result, a large number of test actions generated for apps Connectbot and Omninote attempt to select an item from an empty list or click on an invisible element. Such problems, however, would not occur with ADAMANT. In E-IFML models, we can easily describe preconditions of such test actions using Expressions, so that list item selection is only actioned when the corresponding list is not empty.

Table 4 also lists the percentage of statements that are exclusively covered by each tool (%U). MONKEY, ANDROIDRIPPER, and GATOR achieved an average of 2.8%, 1.8%, and 0.3% in this measure. ADAMANT achieved 21.8%, i.e., 7.8 times as much as MONKEY, 12.1 times as much as ANDROIDRIPPER, and 72.7 times as much as GATOR. MONKEY and ANDROIDRIPPER achieved unique statements coverage over 4.0% on apps Connectbot and Owncloud, because the subjects simplified or omitted some functions when building the E-IFML models for the apps. Nevertheless, ADAMANT significantly outperformed the other three tools from the aspect of statements coverage.

Test generation time (T_g) with ADAMANT and GATOR is considerably shorter than that with MONKEY and ANDROIDRIPPER. Such difference is easily understandable, since test generation using the latter two tools involves executing the generated tests, which can be quite time-consuming but also ensures all the generate actions are executable. In comparison, while a significant percentage of test actions generated by GATOR are inexecutable, ADAMANT does not suffer from the same problem, thanks to the guidance provided by the E-IFML models. The overall time cost of applying the tools to test the object apps is of similar magnitude, if both test generation time and test execution time is considered.

Regarding bugs in the objects, only MONKEY helped to discover bug B8, while neither ANDROIDRIPPER nor GATOR detected any bug. In other words, seven bugs (B1 through B7) were only detected by ADAMANT. In this regard, ADAMANT also performs much better than the other three tools.

819 | ADAMANT *significantly outperforms* MONKEY*,* ANDROIDRIPPER*, and* GATOR *in terms of statement coverage achieved and number of bugs discovered.* |

6.5.4. RQ4: Constraint and Solution Caching.

Table 5 shows the time cost of ADAMANT in generating tests for the apps with or without constraint and solution caching enabled. In particular, the table lists for each app and each configuration the total time for test generation (Total) and the time spent in constraint solving using Z3 (Z3) in seconds, as well as the ratio between the two (Z3/Total). With caching disabled, the time for constraint solving accounts for 88.4% of the total test generation time. The high ratio is largely due to frequent invocations to the constraint solver when generating test cases. Some events in the object apps can only be triggered using user inputs, such as text inputs and list item selections, satisfying certain conditions. Although there is only a small number of GUI elements associated with such events in the object apps, many test scripts contain test actions aiming to trigger such events. For the test actions to be executable, related constraints need to be solved and suitable user inputs need to be constructed.

With constraint and solution caching enabled, a 99% reduction of the total Z3 execution time, i.e., from 15281.8 seconds to 119.6 seconds, was achieved, since only combinations of related constraints for each input need to be solved once and just once under such settings. The results suggest that caching enables most of the test generation processes to finish in about 10 minutes.

Table 5: Test generation time in seconds with and without Z3 solution caching.

App	Without Caching (s)			With Caching (s)		
	Total	Z3	Z3/Total (%)	Total	Z3	Z3/Total (%)
Bookdash	202.0	185.0	91.6	14.5	0.8	5.6
ConnectBot	295.7	154.1	52.1	110.1	3.4	3.0
Goodweather	379.4	346.6	91.4	28.9	0.1	0.6
I2p	160.8	129.1	80.3	25.5	0.6	2.4
Kiwix	436.9	400.8	91.7	29.9	2.0	6.7
Lightning	8550.6	7735.0	90.5	634.5	41.3	6.5
Omninotes	1711.3	1514.0	88.5	143.1	3.7	2.5
OwnCloud	5410.0	4718.8	87.2	556.0	66.9	12.0
RingDroid	145.8	97.8	67.1	38.1	0.7	1.9
Talalarmo	0.8	0.6	75.0	0.1	<0.1	42.9
Total	17293.3	15281.8	88.4	1581.6	119.6	7.6

Constraint and solution caching drastically reduces the test generation time with ADAMANT.

6.6. Test Generation Using ADAMANT *versus Manually*

Experiments described above clearly show that, compared with tools like MONKEY, ANDROIDRIPPER, and GATOR, ADAMANT greatly facilitates the effective and efficient generation of test scripts for Android apps. The significant amount of manual effort required for constructing ADAMANT's input E-IFML models, however, may raise the question that why not invest that effort in directly crafting the tests. In view of that, we investigate also the following research question:

- **RQ5:** How does generating tests using ADAMANT compare with crafting the tests manually in terms of their cost-effectiveness ratios?

To address the research question, we conducted a preliminary controlled experiment, where both approaches are applied to a group of Android apps to produce test scripts.

Objects. We select as the objects four apps from Table 1: Bookdash, Goodweather, Ringdroid, and Talalarmo. These four apps are the smallest from the 10 objects used in the previous experiments, and it took the students 2 to 6 hours to model them. We refrained from using larger apps in this controlled experiment since a longer experiment with multiple sessions would be needed to obtain meaningful results on those apps, which, however, will greatly increase the chance that uncontrolled factors, e.g., breaks between the sessions, influence our experimental results.

Subjects. We recruit as our subjects 12 postgraduate students majored in software engineering and with considerable (i.e., between 2 and 5 year) experience in mobile app testing. We did not ask the undergraduate students from the previous experiments to participate in this experiment, since writing tests is no new task for professionals, and for such a task, experienced graduate students perform similarly to industry personnel [26].

Table 6: Results of the controlled experiment to compare Adamant and Manual approaches.

App	Adamant					Manual				
	S-ID	#K	#A	%C	%U	S-ID	#K	#A	%C	%U
Bookdash	S1	16	65	70	3.6	S1	12	96	70	1.9
	S2	13	73	78	5.3	S2	8	58	75	4.5
	S3	17	86	82	8.2	S3	9	89	78	4.9
	Avg.	15	75	76	5.7	Avg.	10	81	75	3.8
Goodweather	S1	17	40	57	3.0	S2	36	159	59	4.9
	S2	14	41	72	5.1	S2	19	95	74	7.4
	S3	16	57	73	6.0	S3	15	116	80	12.9
	Avg.	16	46	67	4.7	Avg.	23	123	71	8.4
Ringdroid	S1	11	27	69	8.9	S1	10	57	54	2.5
	S2	30	85	73	10.3	S2	6	49	70	3.0
	S3	18	49	74	11.1	S3	12	67	70	3.3
	Avg.	20	54	72	10.1	Avg.	9	58	65	2.9
Talalarmo	S1	11	33	87	2.3	S1	9	47	88	1.1
	S2	14	39	90	2.1	S2	9	78	88	1.8
	S3	15	47	90	2.6	S3	17	69	89	1.4
	Avg.	13	40	89	2.3	Avg.	12	65	88	1.4

Setup. The controlled experiment is conducted in two phases. In phase one, the four objects are randomly assigned to the subjects so that each object is tested by exactly three subjects. In this phase, subjects need to first construct the E-IFML model for their assigned apps and then generate tests. In phase two, the four objects are randomly assigned to the subjects again, so that each object is tested by exactly three different subjects than in phase one. In this phase, subjects need to manually prepare test scripts in Robotium format for their assigned apps. Besides, a 4-hour training was provided to all the subjects before the experiment starts, 2 hours for test script writing using Robotium and 2 hours for E-IFML modeling and test generation using Adamant.

At the end of each phase, all the test scripts produced for each object app are collected. At the end of the experiment, we get six test suites for each object app, three generated using Adamant and the other three crafted manually. Since the two test generation approaches were allocated the same amount of time, we compare their cost-effectiveness in terms of their effectiveness.

To avoid imposing too much burden on the subjects, we limit the experiment time on each app in either phase to 3 hours, resulting in a 6-hour test generation time in total for each student. Running the produced test scripts for debugging purposes or coverage information is allowed during both phases to enable quick feedback. Such settings also ensure that the experiment covers both cases where complete E-IFML models can be constructed and cases where only partial E-

IFML models can be built for test generation.

Results. Table 6 lists, for each object app (App), each subject (S-ID), and each test generation approach, the basic measures as explained earlier: the number of test scripts produced (#K), the number of test actions (#A), the statement coverage achieved (%C), and the percentage of statements exclusively covered (%U). Note that, given an object app and a test generation approach, the results from various subjects are listed in increasing order of statement coverage they achieved, and the average of the three results is reported in the table (row Avg.). Also note that, since no bug was detected by either approach, partly due to the limited time we allocate for conducting the experiment, we do not report the number of bugs detected (#B) in the table.

The two approaches achieved very close average *statement coverage* on apps Bookdash and Talalarmo. We conjecture the reason to be that the two apps are relatively small so that most parts of the apps can be tested in 3 hours using either approach. On average, ADAMANT produced slightly higher coverage (72%) than the manual approach (65%) on Ringdroid. A closer look at the app reveals that Ringdroid contains several complex activities with many GUI elements. For instance, one activity in Ringdroid contains 16 GUI widgets, on which 16 events could be triggered, transiting the app to 4 dialog screens with another 16 element. In such a case, once an E-IFML model has been built for those activities, ADAMANT will traverse the model to generate test scripts exercising the app in a systematic way, while it is more likely for a tester to miss some possible behaviors during the tedious process of manual test script construction. As for GoodWeather, the app is the largest object used in this experiment and it contains the largest number of activities. As a result, all the three subjects failed to model all the activities within the given time duration, which led to a slightly lower coverage by test scripts generated using ADAMANT, compared with the manually constructed ones. The differences between the two approaches in terms of %U are in line with those in terms of %C.

Overall, the experimental results suggest that both test generation techniques are able to produce tests with high coverage: test generation using E-IFML and ADAMANT is slightly better at thoroughly testing parts of an app, while it is relatively easier for manual test construction to quickly explore different behaviors involving a wide range of components of an app.

> *Test generation using* ADAMANT *and manually are comparably effective in terms of statement coverage achieved.*

While the two approaches seem to have comparable effectiveness in statement coverage, we argue there is more to the value of E-IFML models for Android apps than just in test generation. On the one hand, the E-IFML models can be constructed at an early stage (e.g., during detailed design) in mobile development, so that they can benefit not only testers but also developers of the apps [27], while GUI test scripts are typically only produced and utilized when significant effort has been invested in implementing the apps. On the other hand, the cost for maintaining GUI test scripts when the app evolves can be so high that engineers would rather write new tests than to update the old ones [28]. Models, however, can be updated at a relatively smaller price, since they involve fewer low level details. Besides, models have been used to facilitate the automated maintenance of GUI tests [29, 30]. In the future, we also plan to systematically investigate the utilization of E-IFML models for the purpose of GUI test maintenance.

6.7. Threats to Validity

In this section, we discuss possible threats to the validity of our findings in the experiments and how we mitigate them.

6.7.1. Construct validity

Threats to construct validity mainly concerns whether the measurements used in the experiment reflect real-world situations.

An important goal for Android app testing is to find bugs in object apps. In this work, we consider test action executions that terminate prematurely or hang as bug revealing, and we manually check the executions that hang to find out the underlying reasons. While most bug revealing actions indicate real bugs in the apps, we might have missed actions whose execution terminated normally but deviating from the expected behavior. To partially solve that problem, next we plan to develop techniques to automatically detect mismatches between actual execution traces and expected paths on E-IFML models of the generated test scripts.

To measure the quality of generated test scripts, we used the percentage of statements that the tests cover. While statement coverage is one of the most recognised metrics for measuring test adequacy, measures based on other metrics may render the experimental results differently. In the future, we plan to do more experiments using a larger collection of metrics to get a more comprehensive understanding of the performance of Adamant.

6.7.2. Internal validity

Threats to internal validity are mainly concerned with the uncontrolled factors that may have also contributed to the experimental results.

In our experiments, one major threat to internal validity lies in the possible faults in the models we construct for the object apps or in the implementation of the Adamant tool. To address the threat, we provide training to students preparing the E-IFML models and review our models and the tool implementation to ensure their correctness.

The short duration of our controlled experiment, as described in Section 6.6, poses a threat to the validity of our findings regarding the two approaches' capabilities, therefore we refrained from drawing any conclusions quantitatively. To mitigate the threat, the subjects should be allowed to work on the assigned tasks in multiple sessions and for a longer duration, mimicking the settings of real-world modeling processes. We leave such an experiment for future work.

6.7.3. External validity

Threats to external validity are mainly concerned with whether the findings in our experiment are generalisable for other situations.

ADAMANT aims to automatically generate test scripts for Android apps, and we used 10 real-world Android apps in our experiments to evaluate the performance of ADAMANT. While the apps are from different categories and of different sizes, they are all open source apps and the total number of object apps is relatively small. These apps may not be good representatives of the other Android apps, which poses a major threat to the external validity of our findings. In the future, we plan to carry out more extensive experiments on more diversified Android apps to confirm the effectiveness of our technique and tool.

Another threat has to do with the students involved in the experiments. Due to difficulties in recruiting professionals to participate in the experiments, we selected students from appropriate backgrounds as our subjects. While previous studies suggest these students behave similarly to professionals in conducting the tasks assigned to them, extensive experiments involving professional programmers/engineers are needed to better support the external validity of ours findings, e.g., in the context of mobile development in industry.

6.8. Discussion

We discuss in this section lessons learned from this research, limitations in E-IFML and the ADAMANT approach, and future directions for research.

We have gathered several important lessons from this research. The first lesson is that models encoding valuable human knowledge about the apps under consideration really make a difference in GUI test generation, while not using any models or using models of low quality can significantly degrade the generation results. To be the most helpful for the test generation process, the models need to capture not only the properties and actions of elements on an app's GUI but also parts of the app's business logic that are related to the GUI. Without information about the business logic, the models will have only limited power in guiding test generation to explore the more involved app behaviors. The second (related) lesson is that to make the model-driven approach more accessible to practitioners, it is critical to reduce the difficulties in, and the cost of, constructing high-quality models. To that end, an easy-to-use and expressive-enough modeling language can be of great value, while techniques and tools that can effectively help improve the models automatically extracted from apps would also be highly appreciated. Our extension to IFML, as described in this work, constitutes an effort to define such a modeling language. The third lesson we learn is that model-based testing is not necessarily more expensive than manual test preparation. Both techniques have their own areas of excellence and can be used together to best suit the apps under testing.

The experimental results reveal two major limitations of E-IFML and the ADAMANT approach to GUI test generation in their current state. First, no mechanism is provided to help verify the correctness of E-IFML models w.r.t. their corresponding apps. The correctness of the input models is of course extremely important for the generation of quality tests with ADAMANT, but ADAMANT simply assumes at the moment that the input models faithfully reflect the characteristics of the apps from a user's perspective. Second, E-IFML offers no construct to support the specification of test oracles. As a result, all the generated tests essentially resort to the primitive oracle that none of the executable test actions should cause an app to crash or hang. While such oracle managed to help ADAMANT discover interesting bugs in the experiments, it is just weak specification and may leave many unexpected behaviors undetected.

In the future, besides of improving both E-IFML and ADAMANT and overcoming the above-mentioned limitations, we also plan to develop new techniques to make the results of generated tests easier to consume. For instance, one problem worth investigating is how to locate the problems when a generated test fails. Here, a failure can be caused by a bug in the app, a discrepancy in the E-IFML model, or both.

7. Related Work

This section reviews recent works on mobile and GUI interaction modeling and mobile testing that are closely related to this paper.

7.1. Mobile and GUI Interaction Modeling

Model driven engineering (MDE) has been widely used in all stages of software development. In recent years, due to the rapid growth of mobile devices and applications as well as the unique features (e.g., Android fragmentation, short time-to-market, and quick technological innovations) of mobile development, many model-based development (MDD) methods and tools were adapted for mobile platforms. Parada et al. [31] propose an approach that uses standard UML class and sequence diagrams to describe the application structural and behavioral views, respectively, and generates Android code based on the diagrams. Balagtas-Fernandez and Hussmann [32] present a prototype tool named MOBIA to facilitate the development of high-level models for mobile applications and the transformation of those models to platform specific code. Heitkötter et al. [33] propose the MD^2 approach for model-driven cross-platform development of apps. A domain specific textual language is used in MD^2 to define platform independent models for apps, which are then compiled to native projects on Android or iOS. Christoph Rieger [34] proposes a domain specific language named MAML for mobile application development. MAML targets non-technical users and can be used to jointly model data, views, business logic, and user interactions of mobile apps from a process perspective. Moreover, models in MAML can be automatically transformed to generate apps for multiple platforms. These approaches mainly focus on the business modeling for mobile apps.

As GUIs are getting more complex, graphical modeling languages that can visually reflect the detailed GUI interactions are needed. Researches on modeling with IFML are thus emerging [35, 36]. Raneburger et al. [37] examine the usefulness of IFML in multi-device GUI generation, which involves first creating a platform-independent model and then transforming the model to get GUIs for various platforms. Frajták et al. [38, 39] model web applications with IFML, transform the models into their front-end test models, and generate test cases for automatic front-end testing. Their technique focuses on scenario—rather than whole application—modeling and testing, and supports only a limited subset of events (such as clicking and form submitting) and view elements (such as lists and forms). Brambilla et al. [13] extend IFML to support the modeling of simple GUI elements, like containers, components, actions, and events, to facilitate the generation of web views coded in HTML5, CSS3, and JavaScript for mobile apps.

Few existing research work on GUI modeling investigated the use of IFML to facilitate test case generation for Android apps, and, due to features of mobile/Android app GUIs, existing model-based testing methods and tools are unlikely to be as effective if applied directly on mobile apps. In this work, we extend IFML with the support for modeling all important aspects of concrete user interactions with Android apps, and use E-IFML models to guide effective automated test script generation.

7.2. Automated Mobile Testing

Automated testing has long been an important topic in mobile development. In recent years, several successful tools, such as Robotium [18], Appium [40], and MonkeyRunner [41], have been developed for automatically executing test scripts on mobile apps. Meanwhile, many approaches have been proposed for the automatic generation of test scripts. Machiry et al. [42] propose the Dynodroid approach that infers representative sequences of GUI and system events for apps and performs fuzz testing with improved random strategies. Since then, random-based automatic testing approaches have bloomed [43, 3, 44]. Another large body of researches focused on testing mobile applications based on program analysis. Mirzaei et al. [45] present an approach called TrimDroid, which relies on program analysis to extract formal specifications of apps and reduce equivalent user inputs. Anand et al. [46] and Mirzaei et al. [47] employ symbolic execution techniques to systematically generate test inputs to achieve high code coverage on mobile apps.

Model-based testing (MBT) methods and tools have also been developed to generate and execute tests for mobile apps. A large body of other research uses dynamic exploration to build models. Representatives of such works include, e.g., AndroidRipper [10] and its descendant MobiGUITAR [48], both of which are based on GUI ripping [49]. AndroidRipper dynamically analyses an app's GUI and systematically traverses the GUI to construct sequences of fireable events as executable test scripts, while MobiGUITAR constructs a state machine model of the GUI and utilizes the model and test adequacy criteria to guide the generation of test scripts. Su et al. [50] introduce Stoat, an automated model-based testing approach that generates a stochastic model based on Gibbs sampling from an app and leverages dynamic analysis to explore the app's behaviours. Yang et al. [9] propose a grey-box approach that employs static analysis to extract events from an app and implements dynamic crawling to reverse-engineer a model of the app by triggering the events on the running app. While these approaches have been proved to be useful in producing test suites that achieve significant levels of code coverage, none of them utilizes human knowledge about behaviors of the apps to make test generation more effective.

Jaaskelainen et al. [51, 52] propose an open-source framework named Tema for online GUI testing of mobile apps. Tema automatically generates abstract tests based on manually crafted, platform-independent behavioral models of apps that focus on abstract user actions and app state changes. The abstract tests are then translated to concrete tests by mapping abstract actions to platform-dependent user actions. Li et al. [53] propose the ADAutomation technique that generates test scripts for Android and iOS apps by traversing a user provided UML activity diagram modeling user behaviors. Amalfitano et al. [54] propose the juGULAR interactive technique that leverages recorded sequences of user events to facilitate the testing of GUIs that can only "be solicited by specific user input event sequences", or gate GUIs. In comparison, we extend the IFML to support the easy and expressive modeling of Android apps and use E-IFML models to guide the automated test generation for Android apps with Adamant. Experimental results show that test scripts generated by Adamant can achieve higher code coverage and detect real bugs.

8. Conclusion

We present in this paper the Adamant approach to automated Android testing based on E-IFML models. E-IFML is tailored to support easy and expressive Android app modeling. Implementing a path exploration algorithm augmented with constraint solving, Adamant can automatically and effectively process E-IFML models and generate test scripts for Android apps.

We conducted experiments on 10 open-source Android apps to evaluate the performance of Adamant. The results show that Adamant is highly effective in terms of code coverage achieved and the number of bugs detected, and that Adamant significantly outperforms other state-of-the-art test generation tools like Monkey, AndroidRipper, and Gator. Such results confirm that the incorporation of human knowledge into automated techniques can drastically improve the effectiveness of test generation for Android apps.

References

[1] G. de Cleva Farto, A. T. Endo, Evaluating the Model-Based Testing Approach in the Context of Mobile Applications, Electronic Notes in Theoretical Computer Science 314 (C) (2015) 3–21.

[2] H. Muccini, A. Di Francesco, P. Esposito, Software testing of mobile applications: Challenges and future research directions, in: Proceedings of the 7th International Workshop on Automation of Software Test, IEEE Press, 2012, pp. 29–35.

[3] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, A. M. Memon, Exploiting the saturation effect in automatic random testing of android applications, in: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems, IEEE Press, 2015, pp. 33–43.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. M. Memon, MobiGUITAR: Automated Model-Based Testing of Mobile Apps., IEEE Software () 32 (5) (2015) 53–59.

[5] S. R. Choudhary, A. Gorla, A. Orso, Automated Test Input Generation for Android: Are We There Yet? (E), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 429–440.

[6] Google, The Monkey UI android testing tool, `https://developer.android.com/studio/test/monkey.html`.

[7] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, T. Xie, Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 987–992.

[8] M. Brambilla, P. Fraternali, The Interaction Flow Modeling Language (IFML), Tech. rep., version 1.0. Object Management Group (OMG), http://www.ifml.org (2014).

[9] W. Yang, M. R. Prasad, T. Xie, A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications, in: Hybrid Systems: Computation and Control, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 250–265.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, A. M. Memon, Using GUI ripping for automated testing of Android applications, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 258–261.

[11] S. Yang, D. Yan, H. Wu, Y. Wang, A. Rountev, Static control-flow analysis of user-driven callbacks in Android applications, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 89–99.
URL `http://dl.acm.org/citation.cfm?id=2818754.2818768`

[12] J. Whittle, J. Hutchinson, M. Rouncefield, The state of practice in model-driven engineering, IEEE software 31 (3) (2013) 79–85.

[13] M. Brambilla, A. Mauri, E. Umuhoza, Extending the Interaction Flow Modeling Language (IFML) for model driven development of mobile applications front end, in: International Conference on Mobile Web and Information Systems, Springer, 2014, pp. 176–191.

[14] B. Hailpern, P. Tarr, Model-driven development: the good, the bad, and the ugly, IBM Systems Journal 45 (3) (2006) 451–461.

[15] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-integrated development of embedded software, Proceedings of the IEEE 91 (1) (2003) 145–164.

[16] V. Y. Marland, H.-K. Kim, Model-driven development of mobile applications allowing role-driven variants, in: International Conference on Applied Computing and Information Technology, Springer, 2018, pp. 14–26.

[17] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

[18] Github.RobotiumTech, Android UI Testing Robotium, `https://github.com/RobotiumTech/robotium`.

[19] Sirius, The easiest way to get your own Modeling Tool, `https://www.eclipse.org/sirius/`.

[20] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.

[21] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, A. Rountev, Static window transition graphs for Android, Automated Software Engineering 25 (4) (2018) 833–873.

[22] Wiki, List of free and open-source android applications, `https://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications`.

[23] Github.pcqpcq, open-source-android-apps, `https://github.com/pcqpcq/open-source-android-apps`.

[24] I. Salman, A. T. Misirli, N. Juristo, Are students representatives of professionals in software engineering experiments?, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 666–676.

[25] E. Foundation, Eclipse java development tools (JDT), `https://www.eclipse.org/jdt/`.

[26] P. Runeson, Using students as experiment subjects - an analysis on graduate and freshmen student data, Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering (2003) 95–102.

[27] A. Z. Javed, P. A. Strooper, G. N. Watson, Automated generation of test cases using model-driven architecture, in: Proceedings of the Second International Workshop on Automation of Software Test, IEEE Computer Society, 2007, p. 3.

[28] M. Grechanik, Q. Xie, C. Fu, Maintaining and evolving GUI-directed test scripts, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 408–418.

[29] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, X. Li, ATOM: automatic maintenance of GUI test scripts for evolving mobile applications, in: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017, 2017, pp. 161–171.

[30] N. Chang, L. Wang, Y. Pei, S. K. Mondal, X. Li, Change-based test script maintenance for Android apps, in: 2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018, 2018, pp. 215–225.

[31] A. G. Parada, L. B. de Brisolara, A model driven approach for android applications development, in: Computing System Engineering (SBESC), 2012 Brazilian Symposium on, IEEE, 2012, pp. 192–197.

[32] F. T. Balagtas-Fernandez, H. Hussmann, Model-driven development of mobile applications, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2008, pp. 509–512.

[33] H. Heitkötter, T. A. Majchrzak, H. Kuchen, Cross-platform model-driven development of mobile applications with MD^2, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013, pp. 526–533.

[34] C. Rieger, Business apps with maml: a model-driven approach to process-oriented mobile app development, in: Proceedings of the Symposium on Applied Computing, ACM, 2017, pp. 1599–1606.

[35] G. Brajnik, S. Harper, Detaching control from data models in model-based generation of user interfaces, in: International Conference on Web Engineering, Springer, 2015, pp. 697–700.

[36] N. Laaz, S. Mbarki, Integrating IFML models and owl ontologies to derive uis web-apps, in: Information Technology for Organizations Development (IT4OD), 2016 International Conference on, IEEE, 2016, pp. 1–6.

[37] D. Raneburger, G. Meixner, M. Brambilla, Platform-independence in model-driven development of graphical user interfaces for multiple devices, in: International Conference on Software Technologies, Springer, 2013, pp. 180–195.

[38] K. Frajták, M. Bureš, I. Jelínek, Transformation of IFML schemas to automated tests, in: Proceedings of the 2015 Conference on research in adaptive and convergent systems, ACM, 2015, pp. 509–511.

[39] K. Frajták, M. Bures, I. Jelínek, Using the Interaction Flow Modelling Language for Generation of Automated Front-End Tests, in: FedCSIS Position Papers, 2015.

[40] J. Foundation, Appium: Mobile App Automation Made Awesome, `https://appium.io`.

[41] Google, monkeyrunner, `https://developer.android.com/studio/test/monkeyrunner/`.

[42] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: An input generation system for android apps, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 224–234.

[43] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, et al., Caiipa: Automated large-scale mobile app testing through contextual fuzzing, in: Proceedings of the 20th annual international conference on Mobile computing and networking, ACM, 2014, pp. 519–530.

[44] W. Song, X. Qian, J. Huang, Ehbdroid: beyond gui testing for android applications, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2017, pp. 27–37.

[45] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, S. Malek, Reducing combinatorics in GUI testing of android applications, in: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, 2016, pp. 559–570.

[46] S. Anand, M. Naik, M. J. Harrold, H. Yang, Automated concolic testing of smartphone apps, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, 2012, p. 59.

[47] N. Mirzaei, H. Bagheri, R. Mahmood, S. Malek, Sig-droid: Automated system input generation for android applications, in: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2015, pp. 461–471.

[48] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. M. Memon, MobiGUITAR: Automated model-based testing of mobile apps, IEEE Software 32 (5) (2015) 53–59.

[49] A. M. Memon, I. Banerjee, A. Nagarajan, GUI ripping: Reverse engineering of graphical user interfaces for testing., in: WCRE, Vol. 3, 2003, p. 260.

[50] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su, Guided, stochastic model-based GUI testing of Android apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 245–256.

[51] A. Jaaskelainen, M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, T. Takala, H. Virtanen, Automatic gui test generation for smartphone applications - an evaluation, in: 2009 31st International Conference on Software Engineering - Companion Volume, 2009, pp. 112–122.

[52] A. Nieminen, A. Jaaskelainen, H. Virtanen, M. Katara, A comparison of test generation algorithms for testing application interactions, in: 2011 11th International Conference on Quality Software, 2011, pp. 131–140.

[53] A. Li, Z. Qin, M. Chen, J. Liu, ADAutomation: An activity diagram based automated GUI testing framework for smartphone applications, in: Software Security and Reliability, 2014 Eighth International Conference on, IEEE, 2014, pp. 68–77.

[54] D. Amalfitano, V. Riccio, N. Amatucci, V. De Simone, A. R. Fasolino, Combining automated GUI exploration of android apps with capture and replay through machine learning, Information and Software Technology 105 (2019) 95–116.