Automated, Cost-effective, and Update-driven App Testing

CHANH DUC NGO and FABRIZIO PASTORE, SnT Centre, University of Luxembourg, Luxembourg
LIONEL BRIAND, SnT Centre, University of Luxembourg, Luxembourg and School of EECS, University of Ottawa, Canada

Apps' pervasive role in our society led to the definition of test automation approaches to ensure their dependability. However, state-of-the-art approaches tend to generate large numbers of test inputs and are unlikely to achieve more than 50% method coverage.

In this article, we propose a strategy to achieve significantly higher coverage of the code affected by updates with a much smaller number of test inputs, thus alleviating the test oracle problem.

More specifically, we present ATUA, a model-based approach that synthesizes App models with static analysis, integrates a dynamically refined state abstraction function and combines complementary testing strategies, including (1) coverage of the model structure, (2) coverage of the App code, (3) random exploration, and (4) coverage of dependencies identified through information retrieval. Its model-based strategy enables ATUA to generate a small set of inputs that exercise only the code affected by the updates. In turn, this makes common test oracle solutions more cost-effective, as they tend to involve human effort.

A large empirical evaluation, conducted with 72 App versions belonging to nine popular Android Apps, has shown that ATUA is more effective and less effort-intensive than state-of-the-art approaches when testing App updates.

CCS Concepts: • **Software and its engineering → Software verification and validation;**

Additional Key Words and Phrases: Android testing, regression testing, upgrade testing, model-based testing, information retrieval

ACM Reference format:
Chanh Duc Ngo, Fabrizio Pastore, and Lionel Briand. 2022. Automated, Cost-effective, and Update-driven App Testing. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 61 (July 2022), 51 pages.
https://doi.org/10.1145/3502297

1 INTRODUCTION

The business-critical role played by software applications for mobile devices (Apps) in our society [17] has led to the development of dedicated techniques for their automated testing [41]. Since

This project has received funding from Huawei Technologies Co., Ltd, China, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277). Experiments presented in this article were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Authors' addresses: C. D. Ngo and F. Pastore, SnT Centre, University of Luxembourg, JFK 29, Luxembourg, Luxembourg; emails: chanh-duc.ngo@uni.lu, fabrizio.pastore@uni.lu; L. Briand, SnT Centre, University of Luxembourg, JFK 29, Luxembourg, Luxembourg, School of EECS, University of Ottawa, Ottawa, Canada; email: lionel.briand@uni.lu.

1049-331X/2022/07-ART61 $15.00
https://doi.org/10.1145/3502297

most of the code in an App concerns the handling of input values and events, test automation approaches automatically generate sequences of events and input values (hereafter, input sequences) that simulate the use of the App under test in its deployed environment. These approaches mainly differ with respect to the strategy used to create input sequences, such as random, evolutionary, and model-based approaches relying either on static or dynamic information [41].

Unfortunately, state-of-the-art automated App testing techniques show limited code coverage capabilities, thus indicating they are unlikely to exercise all the features of the App under test. For example, they typically exercise about half of the methods implemented by commercial apps [70]. As a result, all methods and instructions that are not automatically tested should be exercised by manually implemented test cases, an expensive task that may delay the App release. Also, though existing techniques show a degree of complementarity [70], state-of-the-art approaches do not attempt to integrate them to achieve better results.

Existing testing approaches do not account for the high release frequency of a typical App's lifecycle, which are usually driven by marketing strategies aiming at increasing visibility [13, 22, 44]. As a result, existing work does not include effective means of prioritizing the testing of modified or newly introduced features and are thus not addressing one of the major needs of App developers. However, this is an important requirement for any testing strategy, as exercising all the features of an App in each release is enormously wasteful. Existing work on testing App upgrades is limited to the selection of subsets of events that may trigger modified code [61] or the selection of regression test cases [15]. This is, however, not adequate when, to start with, available test cases do not exercise all the new and modified features of the software.

Finally, the current body of work does not address the *oracle problem* [11, 41, 57]. More precisely, testing techniques cannot discover functional failures beyond crashes and the manual verification of the App outputs is difficult due the large number of inputs they exercise [15]. However, in the context of frequent App updates, with a test input generation strategy that effectively exercises updated features, it is conceivable to address the oracle problem by relying on dedicated strategies to minimize test inputs. Failures affecting unchanged features (i.e., regressions) can be automatically detected by comparing the output of different App versions for a same input [26, 37, 51, 60], whereas the output of new and modified features can instead be verified, at reduced costs, by relying on internal or external crowdsourcing [49]. Nevertheless, such solutions are only practical if the number of test inputs is kept down to a reasonable number.

Keeping the number of test inputs to the strict minimum is important to minimize human intervention, since it may be required when executing the same inputs on different software versions, e.g., to adapt input sequences to changes in the GUI [40, 47]. Further, screenshots of the results must be visualized after every input. Unfortunately, state-of-the-art App testing approaches generate large test suites, while test suite reduction approaches require to perform runtime monitoring of the App, which slows down execution and diminishes test automation effectiveness [15].

In summary, to address the limitations above, we aim to achieve the following two objectives: (O1) maximize the number of updated methods and their instructions that are automatically exercised within practical test execution time, and (O2) generate a significantly reduced set of inputs, compared to state-of-art approaches, thus decreasing human effort.

To achieve the two objectives above, it is necessary to integrate multiple analysis strategies. Objective O1 can be effectively achieved by means of static analysis, to determine updated features (e.g., through the identification of updated methods [61]) and identify the inputs that may trigger a specific feature (e.g., the input that leads to a particular Window) [77]. Unfortunately, static analysis alone may not enable the effective testing of Apps; indeed, they typically rely on APIs dedicated to input handling that are hardly processed by static analysis tools, as discussed in related work [57]. Random exploration is thus required to discover, at runtime, inputs that may trigger a potentially large subset of modified methods. Unfortunately, random exploration might be particularly inefficient and conflict with objective O2 (e.g., it may require thousands of inputs to exercise features that depend on specific App states). For this reason it is necessary to determine which inputs bring the App into distinct program states by relying on dynamically refined state abstraction functions [29] and by identifying dependencies among App features (e.g., to determine that an option in the settings page enables a specific feature).

In this article, we present **ATUA (Automated Testing of Updates for Apps),**[1] the first approach that integrates multiple test strategies to efficiently use the test budget and achieve the two above-mentioned objectives. The rationale followed by ATUA is that Apps can be cost-effectively tested by combining static and dynamic program analysis to select the inputs that exercise updated methods, our test targets. Also, given the complexity of Apps, testing should be performed incrementally, by focusing first on objectives that are easier to achieve. For this reason, ATUA works in three phases: (1) it exercises all the features that may trigger modified methods (e.g., submitting a registration form that is processed by an updated method), (2) to maximize coverage in the presence of data-dependencies, it exercises updated features with diverse input values (e.g., a diverse set of values in a form), (3) to maximize coverage in the presence of state-dependencies, it exercises related features (e.g., submit a registration form after changing language settings). ATUA implements a model-based approach that integrates a *dynamically refined state abstraction function* and complementary testing strategies, including (1) coverage of the *model structure*, (2) coverage of the *App code*, (3) *random* exploration, and (4) coverage of *dependencies* among App windows.

ATUA generates models of the App under test by combining static and dynamic program analysis. It extends static program analysis approaches [77] to automatically generate *extended window transition graphs (EWTG)*, i.e., finite state machines that capture which inputs trigger window transitions and updated methods. Also, it introduces a state abstraction function that refines the states of the EWTGs to capture differences in the user interface that are not detected by means of static analysis (e.g., the presence of dynamically disabled buttons). The state abstraction function is automatically refined to eliminate or, when not possible, reduce non-determinism while minimizing the number of abstract states.

To automatically exercise Apps, ATUA relies on the generated EWTGs to identify the sequences of inputs that trigger the execution of updated methods. When there are discrepancies between the EWTGs and the observed behavior, random exploration is used to refine the former. Code coverage is used to identify the methods that require additional testing effort. Finally, using information retrieval techniques [66], ATUA identifies dependencies between App windows that may prevent the execution of certain methods.

We assume that, for every software version, engineers are interested in testing the updated methods only. However, the general principles behind ATUA can easily be adopted also for other ways of characterizing change, e.g., based on impact analysis [58]. Indeed, other criteria for selecting target methods are straightforward to integrate into ATUA.

An empirical evaluation conducted with nine popular, commercial Apps shows that, compared to state-of-the-art approaches (i.e., DM2 [12], APE [29], and Monkey [5]), ATUA leads to reduced test costs. Indeed, it generates less than 70%, 4%, and 2% of the inputs generated by DM2, APE, and Monkey, respectively. By automatically exercising, on average, 2.6 instructions belonging to updated methods for every generated test inputs, ATUA is the most cost-effective approach. Further, on average, ATUA, for a same test execution budget (e.g., one-hour test execution time), improves the method and instruction coverage achieved by the second best, state-of-the-art approach by at least 10%.

[1] Atua is also the name of spirits in Polynesia, https://maoridictionary.co.nz/word/494.

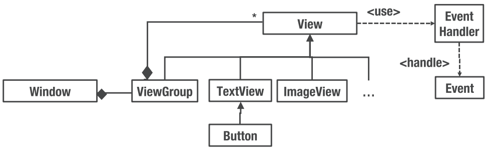
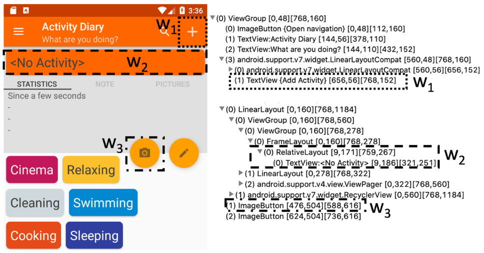


Fig. 1. Overview of the Android Apps's GUI architectural components.



```
▼(0) ViewGroup [0,48][768,160]
    (0) ImageButton {Open navigation} [0,48][112,160]
    (1) TextView:Activity Diary [144,56][378,110]
    (2) TextView:What are you doing? [144,110][432,152]
  ▼(3) android.support.v7.widget.LinearLayoutCompat [560,48][768,160]
    ▶(0) android.support.v7.widget.LinearLayoutCompat [560,56][656,152]
     (1) TextView {Add Activity} [656,56][768,152]   W1

  ▼(0) LinearLayout [0,160][768,1184]
    ▼(0) ViewGroup [0,160][768,560]
      ▼(0) ViewGroup [0,160][768,278]
        ▼(0) FrameLayout [0,160][768,278]
          ▼(0) RelativeLayout [9,171][759,267]
              (0) TextView:<No Activity> [9,186][321,251]   W2
       ▶(1) LinearLayout [0,278][768,322]
       ▶(2) android.support.v4.view.ViewPager [0,322][768,560]
    ▶(1) android.support.v7.widget.RecyclerView [0,560][768,1184]
    (1) ImageButton [476,504][588,616]   W3
    (2) ImageButton [624,504][736,616]
```

Fig. 2. Example of a GUITree. We use different dotted boxes to match widgets in the tree to the pixels in the screen.

The article is structured as follows: Section 2 introduces background technologies. Section 3 provides the technical details of the proposed approach. Section 4 reports on the results of our empirical evaluation. Section 5 discusses related work. Section 6 concludes the article.

2 BACKGROUND

2.1 App Design and Architecture

In this article, we target Android Apps, since Android is the most adopted platform and is widely investigated in research [22]. However, most of the solutions proposed here should be easily tailorable to other platforms.

When an App is running, the end-user interacts with the active *Window* of the App (i.e., the Window being rendered on the screen). Windows consist of a hierarchy tree of widgets; in this article, we use the term *GUITree* to refer to such hierarchy trees [9, 29, 65, 79].[2]

A widget extends the class *View*. Figure 1 shows a portion of the hierarchy tree of the class *View*. Figure 2 shows a portion of the GUITree for a window of Activity Diary, one of our case study Apps (see Section 4).

Each widget has a set of properties associated to it. Widgets can be associated to *EventHandlers* that are invoked by the OS when specific *InputEvents* are triggered by the end-user. Typical InputEvents include click, long click, swipe, and keypress.

In Android, the application logic is typically implemented by Activity classes that are instantiated by the framework and act as controllers of the Model-View-Controller design pattern [52]. Inter-process communication, instead, is managed by the *Intent* resolution mechanism [3]. More precisely, in Android, a system event (e.g., indicating a battery being low) or a message exchanged between apps (e.g., a URL sent by the browser to a music player App) is referred to as an *Intent*. To handle Intents, an App declares in its XML configuration file an Activity that the OS will instantiate and execute when a specific Intent type should be received by the App.

2.2 App Testing Automation

System-level testing of an App through its GUI (i.e., GUI testing) is performed through sequences of *test inputs* that can be either Events or Intents. Functional GUI testing aims at exercising (i.e., render active) all declared Windows, triggering all event handlers, and covering all the code of the App under test. App testing automation aims to generate input sequences to achieve these objectives at the lowest cost possible.

For a complete overview of App testing automation approaches, the reader is referred to recent surveys [41, 67]. *Model-based* solutions are the most commonly ones reported in the literature [67].

[2] Other work uses the term GUITree to refer to the sequence of App windows encountered during testing [1].

In model-based testing approaches, the model used to drive testing is typically a **finite state machine (FSM)**. It can be formally described as a tuple $(S, A, T, \mathcal{L})$ [29], where

- S is a set of states.
- A is a set of actions.
- T is a set of state transitions. Each transition has a source state and a target state. It is triggered by an action $\alpha \epsilon A$.
- $\mathcal{L}$ is an abstraction function, which might be used to: (1) assign a Window to a state and (2) match an InputEvent or an Intent to an action.

Model-based approaches differ regarding the type of analysis adopted to identify states and transitions (i.e., dynamic [12, 65] or static [77, 78]), the abstraction functions used (i.e., predefined [12, 65] or adaptable [29]), and the model exploration strategies they rely on (i.e., offline [65] or online [12]). Adaptable state abstraction functions have been shown to lead to more effective App testing [29] but they have never been used in testing frameworks that enable the effective combination of static and dynamic analysis. Further, existing model-based approaches do not prioritize the testing of updated methods, which is our objective here, and thus, we require dedicated input generation algorithms. To leverage the benefits of static and dynamic program analysis, ATUA integrates (1) Gator, a tool that statically identifies states and transitions, (2) DroidMate2 (DM2), a model-based framework that performs online testing, dynamic identification of states and transitions, and enables the combination of static and dynamic analysis, (3) a dedicated algorithm for test input generation, and (4) a custom and adaptable state abstraction function.

DM2 [12] consists of two main engines for exploration and automation, respectively. The former drives the interaction with the App and derives the model of the App under test. The exploration is driven by a set of user-defined strategies. The latter translates actions into concrete commands on the device. In *DM2*, an App state is univocally identified by the set of UI elements in the user interface and their state-related properties. UI elements are identified by means of their descriptive ID or their image bytes cut from a screenshot. State-related properties are given by the position of the elements and other widget specific characteristics. ATUA relies on the *DM2* automation engine, which provides an API to send inputs to the App under test and retrieve information about the displayed UI elements. In ATUA, App exploration is driven by a custom algorithm (see Section 3.5) that relies on statically derived models and a custom state abstraction function.

Gator [77, 78] is a static analysis tool that creates models of the App under test in the form of *window transition graphs (WTGs)*. WTGs are FSMs representing the possible window sequences and their associated events and callbacks. Though WTGs can be used to enable model-based testing, they have been mostly used to detect resources leaks [75].

3 PROPOSED APPROACH: ATUA

Within an updated App, we can distinguish among existing features (i.e., features present in previous versions of the App under test) and new features (i.e., features introduced in the App under test). Existing features can be unchanged (i.e., their functional requirements did not change), modified (i.e., their functional requirements did change), or repaired (i.e., modified because their implementation did not match its functional requirements).

In our work, we aim to automatically exercise *updated features*, including new, modified, and repaired features. More precisely, we focus on features that are implemented either by introducing new methods or by modifying existing methods.[3] In this article, we use the term *updated methods* to refer to both new and modified methods, which are our test targets.

[3] Based on related work, 81% of the updates concern Java files, while only the remaining 19% concern manifest files (e.g., permissions) or layout declarations in XML files [61].

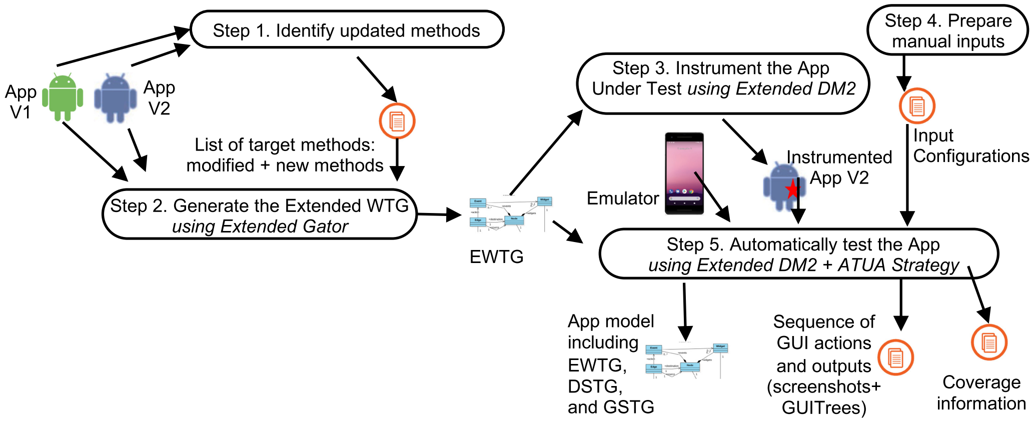

Step 1. Identify updated methods
App V1
App V2
List of target methods: modified + new methods
Step 2. Generate the Extended WTG using Extended Gator
EWTG
Step 3. Instrument the App Under Test using Extended DM2
Emulator
Instrumented App V2
Step 4. Prepare manual inputs
Input Configurations
Step 5. Automatically test the App using Extended DM2 + ATUA Strategy
App model including EWTG, DSTG, and GSTG
Sequence of GUI actions and outputs (screenshots+ GUITrees)
Coverage information

Fig. 3. Overview of the ATUA process to test App updates.

The testing activity performed by ATUA is driven by an App model with the objective of exercising a set of test targets (i.e., updated methods). The App model is initially created by static program analysis procedures and then refined during testing.

The App model metamodel is shown in Figure 4 and described in Section 3.1. It consists of three parts: (1) an **Extended Window Transition Graph (EWTG)**, (2) a **Dynamic State Transition Graph (DSTG)**, and (3) a **GUI State Transition Graph (GSTG)**. The three graphs are FSMs capturing how input values trigger changes in the state of the App under test. The *EWTG* models the sequences of windows being visualized after specific inputs (Events or Intents). For every input, the EWTG keeps trace of the name of the handlers associated to the input and the list of test targets that may be invoked during the execution of the input handler. The *GSTG* is a fine-grained model that captures every visual change in the GUI (e.g., the color of a button) that might be triggered by an action on the GUI. An action is an instance of an input (e.g., click on a specific Button widget). Finally, the *DSTG* models the abstract states of the visualized Windows and the state transitions triggered by events. Abstract states are identified by a state abstraction function to eliminate possible non-determinism. The DSTG plays a critical role to optimize the test budget and identify a reduced set of input events; indeed, it helps determine a correct and reduced sequence of events necessary to reach a specific Window from another one.

Figure 3 provides an overview of the process implemented by ATUA to test App updates. In Step 1, ATUA compares the previous (App V1 in Figure 3) and the updated (App V2) version of the App under test to identify the updated methods. In Step 2, ATUA relies on Soot [56], a static analysis framework, and an extended version of Gator, to generate the EWTG. In Step 3, ATUA relies on DM2 to generate a version of the App that is instrumented to trace code coverage. In Step 4, engineers manually specify test inputs that are unlikely to be generated automatically (e.g., the login credentials for Apps that require a user to be registered on a remote platform). In Step 5, ATUA exercises the App under test by relying on an extended version of DM2 that integrates the ATUA test algorithm. During testing, ATUA refines the App model and relies on it to identify the actions to perform on the GUI. For example, ATUA uses the App model to identify the action that, in the current window, may lead to the execution of a test target.

The main output of Step 5 is the sequence of GUI actions performed during testing and the outputs (i.e., the screenshot of the active Window and the corresponding GUITree) generated by the App under test after every action. This sequence is used by engineers to verify if the behavior of the App is as expected (test oracle). As mentioned earlier, to verify App results, engineers can rely on two complementary state-of-the-art approaches, not addressed by ATUA, that, respectively, target regression failures in unchanged features and failures in newly implemented, repaired, and modified features. To discover regression failures, engineers can replicate, on a previous App version, the test input sequences generated for the updated App and automatically compare the generated outputs. Differences in the outputs generated by the two versions should indicate the presence of a regression fault. To discover failures in new and repaired features, engineers can visualize the GUITrees or the screenshots of the active Window rendered after each Action. The visual inspection of the App outputs enables an engineer to determine the presence of functional failures, based on expected behavior, whether specifications are implicit or documented. For example, the engineer shall determine if the Window rendered after each Action includes the expected content and is well positioned. Also, the engineer shall inspect GUITrees to determine if the widgets within a Window have the expected properties. In Section 4, we discuss to what extent ATUA reduces the cost associated to the manual activities entailed by the test oracle strategies above, with respect to other state-of-the-art test automation solutions.

In addition, ATUA provides, as output of Step 5, an App model including the EWTG, the DSTG, and the GSTG. The App model is generated and continuously refined during testing. Further, it reports coverage information, i.e., the sets of updated methods and instructions belonging to updated methods that have been exercised during testing.

In the following, we provide additional details about the App model metamodel (Section 3.1) and describe Steps 1 to 5 (Sections 3.2 to 3.5), except for Step 3, which is already automated by DM2.

3.1 App Model Metamodel

Figure 4 shows the ATUA metamodel as a UML class diagram. Figure 5 shows an example App model built when testing Activity Diary.

The EWTG is consistent with the WTG generated by Gator. Each WindowTransition is triggered by an *Input*, either an *InputEvent* or an *Intent*. An InputEvent is associated to the Widget that declares its EventHandler. If the EventHandler is not declared by a Widget (e.g., for the event *PressHome*), then the InputEvent is not associated to any target Widget. Each Widget belongs to one Window.

In addition to the concepts captured by the WTG, the EWTG generated by ATUA also captures the list of modified methods that can be triggered by the Input (i.e., the attribute *targetMethods* of class *Input*), which are used to drive testing. A WindowTransition triggered by Inputs with associated *targetMethods* is a *target Transition*. Similarly, a Window that is the source for at least one target Transition is a *target Window*. *TargetMethods* are identified by our Gator extension (see Section 3.3). The EWTG also captures the dialogs and menus that can be opened by an Activity (e.g., association *triggeredDialogs*). Finally, it also models the HiddenHandlers of a Window, which are introduced in Section 3.3.

The GSTG captures the same information provided by the models generated by DM2. The state of a Window is captured by its GUITree, which is a composition of Widgets. For each Widget, we record the values associated to its properties and derive the Widget hash (to associate an ID to the current state of the Widget) according to the DM2 strategy. The hash of the GUITree is then derived from the hash of its Widgets. In addition, the GSTG captures the name of the Activity running when the GUITree is visualized (see attribute *activityName*), which we derive, at runtime, from logcat [4]. The transition between GUITrees is triggered by an Action, which can be handled either by the Widget (*WidgetAction*) or by the visualized Window (*WindowAction*). The enumerations *WidgetActionType* and *WindowActionType* list the type of actions that can be performed to trigger GUITreeTransitions. Actions might have additional information (i.e., actionData)

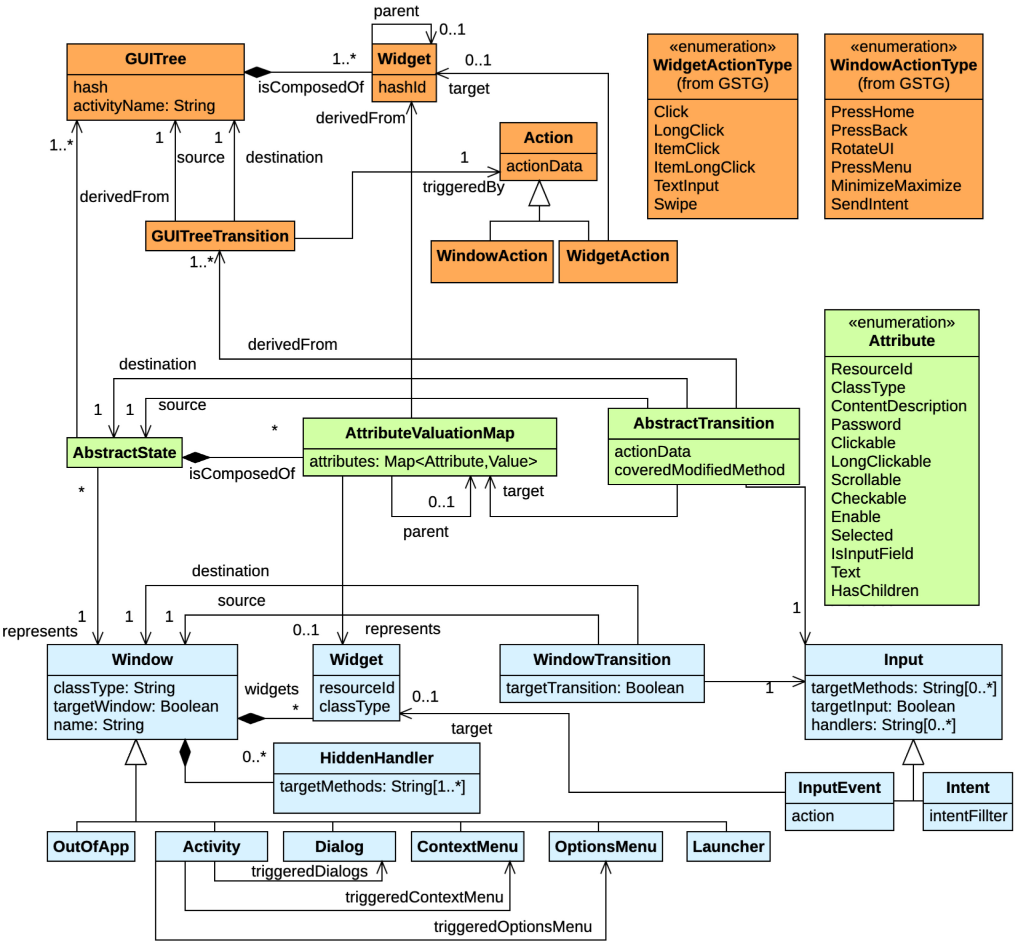
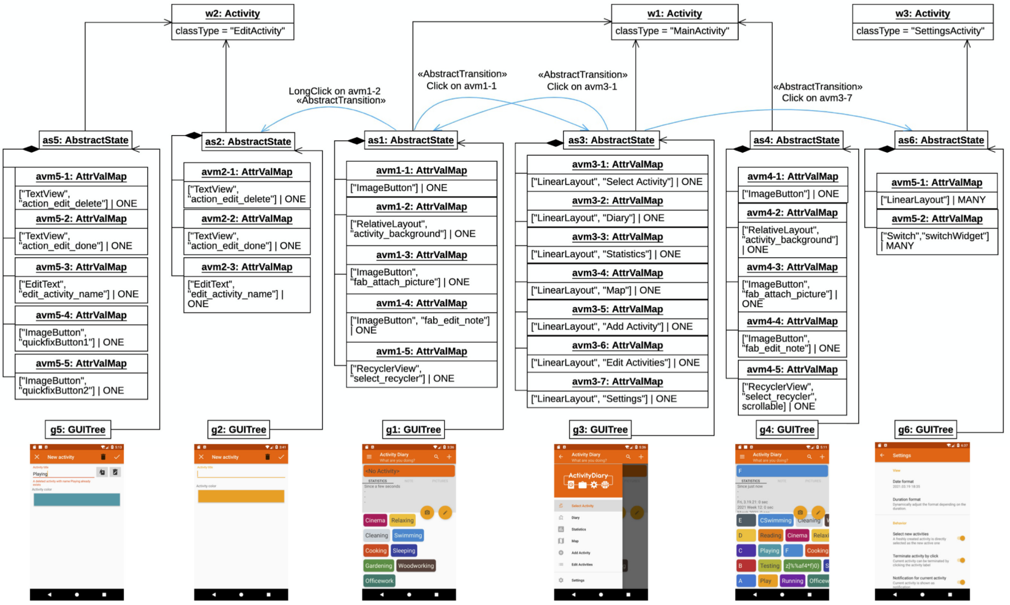


Fig. 4. ATUA metamodel. Colors are used to identify classes belonging to a specific metamodel component: light blue for EWTG (bottom), light green for DSTG (middle), orange for GSTG (top).

associated to them; for example, the text provided to the App under test by a TextInput action or the start and end coordinates of a Swipe action.

The DSTG provides abstract states that group together GUITrees in which a same Action triggers a same App behavior (e.g., leads to a same abstract state). The DSTG enables ATUA to efficiently test the App under test by determining the shortest sequence of Actions that reaches a target Window. Also, abstract states capture the conditions under which a specific Action can trigger a modified method for a certain Window. Abstract states are thus a means to minimize the number of Actions generated by the test automation approach.

Each AbstractState consists of a number of AttributeValuationMaps, each one abstracting the state of the widgets belonging to the GUITree that have the same set of attribute valuations. An *AttributeValuationMap* is a map of pairs ⟨attribute,value⟩. The enumeration *Attribute* in Figure 4 provides the list of attributes appearing in the AttributeValuationMap. The AttributeValuationMap has a cardinality attribute, which indicates how many widgets have the same attribute valuations. For example, in Figure 5, the AbstractState *as6* includes many LinearLayout widgets (*LL* in the figure, cardinality *MANY*), one for each item in the displayed list.


w2: Activity
classType = "EditActivity"
w1: Activity
classType = "MainActivity"
w3: Activity
classType = "SettingsActivity"
«AbstractTransition»
Click on avm1-1
«AbstractTransition»
Click on avm3-1
LongClick on avm1-2
«AbstractTransition»
«AbstractTransition»
Click on avm3-7
as5: AbstractState
as2: AbstractState
as1: AbstractState
as3: AbstractState
as4: AbstractState
as6: AbstractState
avm5-1: AttrValMap
["TextView", "action_edit_delete"] | ONE
avm5-2: AttrValMap
["TextView", "action_edit_done"] | ONE
avm5-3: AttrValMap
["EditText", "edit_activity_name"] | ONE
avm5-4: AttrValMap
["ImageButton", "quickfixButton1"] | ONE
avm5-5: AttrValMap
["ImageButton", "quickfixButton2"] | ONE
avm2-1: AttrValMap
["TextView", "action_edit_delete"] | ONE
avm2-2: AttrValMap
["TextView", "action_edit_done"] | ONE
avm2-3: AttrValMap
["EditText", "edit_activity_name"] | ONE
avm1-1: AttrValMap
["ImageButton"] | ONE
avm1-2: AttrValMap
["RelativeLayout", "activity_background"] | ONE
avm1-3: AttrValMap
["ImageButton", "fab_attach_picture"] | ONE
avm1-4: AttrValMap
["ImageButton", "fab_edit_note"] | ONE
avm1-5: AttrValMap
["RecyclerView", "select_recycler"] | ONE
avm3-1: AttrValMap
["LinearLayout", "Select Activity"] | ONE
avm3-2: AttrValMap
["LinearLayout", "Diary"] | ONE
avm3-3: AttrValMap
["LinearLayout", "Statistics"] | ONE
avm3-4: AttrValMap
["LinearLayout", "Map"] | ONE
avm3-5: AttrValMap
["LinearLayout", "Add Activity"] | ONE
avm3-6: AttrValMap
["LinearLayout", "Edit Activities"] | ONE
avm3-7: AttrValMap
["LinearLayout", "Settings"] | ONE
avm4-1: AttrValMap
["ImageButton"] | ONE
avm4-2: AttrValMap
["RelativeLayout", "activity_background"] | ONE
avm4-3: AttrValMap
["ImageButton", "fab_attach_picture"] | ONE
avm4-4: AttrValMap
["ImageButton", "fab_edit_note"] | ONE
avm4-5: AttrValMap
["RecyclerView", "select_recycler", scrollable] | ONE
avm5-1: AttrValMap
["LinearLayout"] | MANY
avm5-2: AttrValMap
["Switch","switchWidget"] | MANY
g5: GUITree
g2: GUITree
g1: GUITree
g3: GUITree
g4: GUITree
g6: GUITree

Legend: Straight, black arrows capture associations. To avoid cluttering, we rely on curved, blue arrows to model AbstractTransitions. The action and target of abstract transitions are reported in textual form. For illustration purposes, we show screenshots instead of GUITree hashes. Cardinality of AttrbuteValuationMaps are reported after the "|" symbol.

Fig. 5. Example of an App model, represented as a UML object diagram.

Since the DSTG is used to drive testing, i.e., to select the Actions to be triggered at runtime, the AbstractState captures only the attributes of widgets that are *interactive*. A widget is *interactive* when it is enabled, visible, and is an instance of a class that can be the target of any action of type WidgetActionType (see Figure 4).

At runtime, during testing, ATUA identifies AbstractStates through a dedicated *abstraction function* ($\mathcal{L}$). ATUA automatically defines a distinct $\mathcal{L}$ for each Window of the App under test. $\mathcal{L}$ relies on a predefined set of *reducers*, i.e., functions that extract the value of a property of a widget [29]. Table 1 shows the list of reducers implemented by ATUA. Two AbstractStates differ when at least one value differs across their respective AttributeValuationMaps or when they have a different cardinality. For example, in Figure 5, the AbstractStates *as1* and *as3* differ, because *as1* does not contain the AttributeValuationMaps for the LinearLayouts belonging to the drawer menu (to save space, we do not report all the AttributeValuationMaps for *as3*). The AbstractStates *as1* and *as4* are different, because in *as4* the RecyclerView (avm4-5) becomes scrollable.

In the DSTG, state transitions are captured by AbstractTransitions. An AbstractTransition is univocally identified by the *actionType*, its *source*, its *target*, its *destination*, and its *actionData*. The *actionType* matches one of the items belonging to the enumerations WidgetActionType or WindowActionType. The *actionData* is captured only for two actionTypes (i.e., Swipe and Intent) that usually lead to distinct AbstractStates depending on their action data. In the case of Swipe, the actionData indicates the direction of the Swipe action (i.e., Up, Down, Left, Right). For Intents, the actionData matches the Intent input text, because we expect engineers to provide one manual input for each possible Intent type (e.g., one different URL for each of the file types supported by an App). We leave to future work the definition of functions that provide an abstract representation for the data associated to other types of actions (e.g., to distinguish between numeric, alphabetic, or non-alphanumeric data provided to TextInputs).

Table 1. ATUA Reducers

	Reducer	Description
1	R_{RID}	Resource ID.
2	R_{CN}	Class name.
3	R_{CD}	Value of *Content description.*
4	R_P	Value of *Password.*
5	R_C	Value of *Clickable.*
6	R_{LC}	Value of *Long Clickable.*
7	R_S	Value of *Scrollable.*
8	R_{Ch}	Value of *Checked.*
9	R_E	Value of *Enabled.*
10	R_S	Value of *Selected.*
11	R_I	True if it is an input field.
12	R_T	Value of *Text.*
13	R_{HC}	True if the widget contains one or more children.

We indicate the value of the *property* reported by each.

Table 2. Refinement of ATUA State Abstraction Function

Level	Reducers applied to interactable Widget	Reducers applied to interactable Widget Children
L1	R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E, R_P, R_S, R_I, R_C, R_{LC}, R_S	
L2	R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E, R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T	
L3	R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E, R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T, R_{HC}	
L4	R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E, R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T	R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E, R_P, R_S, R_I, R_C, R_{LC}, R_S
L5	R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E, R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T	R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E, R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T

In blue, we show the reducers introduced in finer granularity level.

A DSTG may include non-deterministic AbstractTransitions, that is, transitions with the same actionType, outgoing from the same AbstractState, but reaching different AbstractStates. Non-deterministic AbstractTransitions may prevent us from finding the correct sequence of Inputs necessary to reach the states in which target methods could be triggered. ATUA detects non-determinism at runtime, during testing, when an Action does not bring the App into the expected AbstractState. When this happens, ATUA refines $\mathcal{L}$ for the AbstractState in which the action had been triggered. It does so according to five levels of granularity, which are captured in Table 2. With level L1, $\mathcal{L}$ distinguishes states based on static information about the widgets (i.e., resource ID and class) and information about how they can be interacted with (i.e., reducers appearing in rows 3 to 12 of Table 1). With level L2, in addition to the information accounted for in L1, $\mathcal{L}$ includes the text associated to the widget, which often affects the behavior of an app (e.g., invalid characters in a textbox may prevent a state transition). With level L3, $\mathcal{L}$ also reports the number of children of a widget (i.e., R_{HC}). L3 is useful, because the interactive widgets captured by $\mathcal{L}$ may include non-interactive children whose state is not captured by $\mathcal{L}$ but may characterize the current state (e.g, through descriptive labels). With levels L4 and L5, $\mathcal{L}$ captures, for every interactive widget, the same information as L2 and, in addition, for every child, the information captured by levels L1 and L2, respectively.

Figure 6 shows the result of the refinement of $\mathcal{L}$ for the abstractState *as3*. By applying $\mathcal{L}$ with level L1, all the clickable elements on the Window, which are LinearLayouts with the same properties, except for the text, resulted into a same AttributeValuationMap with cardinality *MANY*. At runtime, ATUA detects non-determinism; indeed, a click on this AttributeValuationMap may lead to two different AbstractStates: *as1* and *as6*. The refinement of $\mathcal{L}$, which leads to level L2, allows ATUA to distinguish all the different clickable elements, since the text property is included into the AttributeValuationMap, thus eliminating non-determinism.

3.2 Step 1. Identify Updated Methods

In the first step, ATUA identifies methods that have been modified or introduced by the new version of the App (i.e., V2 in Figure 3). This task might be accomplished through source code comparison across versions [50]. However, to enable experiments with commercial Apps, we developed a toolset (hereafter, AppDiff) that compares compiled Android Apps.

AppDiff is an extension of *LibScout* [18], a lightweight static analysis tool for Android. It first generates a hashtree over the bytecode for each App. The hashtree is a three-layered Merkle tree

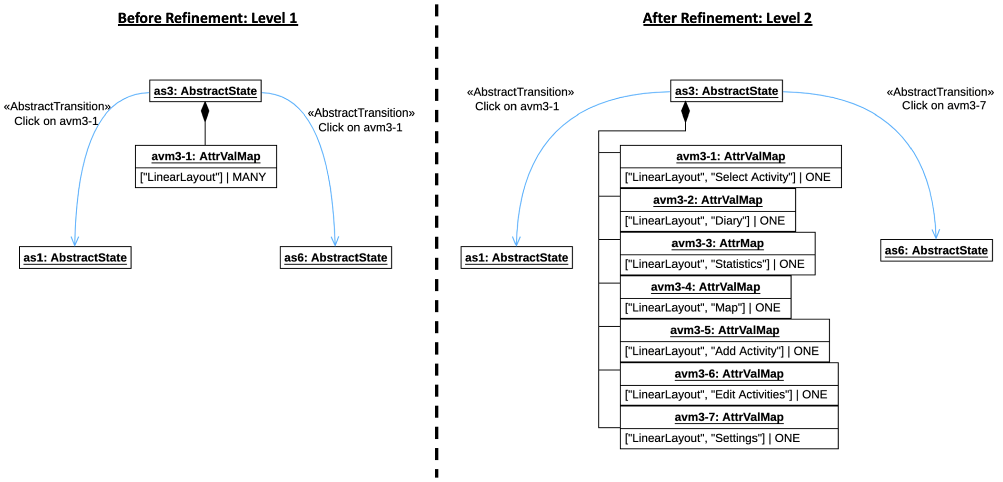


Fig. 6. Example of refinement of $\mathcal{L}$. For the notation used, see the Legend of Figure 5.

in which parent hashes are generated from their child nodes. The three layers model the flattened package structure that is preserved in the compiled code, i.e., packages, classes, and methods.

The tree is built bottom up starting with the method hashes. A method hash is computed over the method signature and the opcodes in bytecode instructions. To identify code-level changes across App versions, we additionally store package, class, and method names along with the hashes. To efficiently check for differences, two hash trees are matched top-down starting with the package hashes. Methods that share the same name but have a different hash have been modified. New methods appear only in the most recent version.

3.3 Step 2. Generate the Extended WTG

ATUA generates the Extended WTG by means of static program analysis; more precisely, by performing, on the updated App, the analysis implemented by an extended version of Gator and Soot.

The original version of Gator works by processing Android bytecode and XML layout files. For the analysis of bytecode, Gator relies on Soot. Bytecode analysis is used to identify the types of Window (i.e., Activity, Dialog, OptionsMenu, and ContextMenu) that are programmatically specified in the App. Bytecode analysis is also used to identify the widgets that compose a window and the associated event handlers. Gator identifies widgets that extend the class *android.view.View* and its handlers. Event handlers' code is processed to determine window transitions. XML layout files are processed to identify additional event handlers.

Our extensions to Gator address some known limitations [39]. More precisely, we support the identification of window transitions triggered by Fragments and RecyclerView, which are widget containers that are not identified by Gator as such. Our extensions associate the contained widgets to the window that declares either the Fragment or the RecyclerView. Also, in the EWTG, we associate each WindowTransition with the Input triggering the transition (the information is provided by Gator).

We rely on Soot to traverse the backward call graph of every updated method *m*. During the traversal, when we encounter a method that has been identified by Gator as an event handler *e*, we update the EWTG to trace the fact that the inputs associated to the WindowTransition triggered by the event handler *e* can lead to the execution of the updated method *m* (i.e., we add the updated

```
 1{
 2  "BookInsertionAndSearch" : {         //input pattern
 3    "Windows" : [   "ACT[bookcatalogue.EditAuthorList]1741", "ACT[bookcatalogue.BookEdit]1802"
      "ACT[bookcatalogue.BookISBNSearch]1843" ],
 4    "DataFields" : {
 5      "isbn" : {
 6        "resourceIdPatterns" : [ "isbn_txt" ]
 7      },
 8      "title" : {
 9        "resourceIdPatterns" : [ "title_txt" ]
10      },
11...
12    },
13    "Instances" : [
14      {
15        "isbn" : "0387284540",
16        "title" : "Applied probability and statistics",
17        "publisher" : "Springer",
18        "pages" : "350",
19        "list_prices" : "69",
20        "format" : "Hard Cover",
21        "genre" : "Unfiction",
22        "language" : "English"
23      }
```

Fig. 7. Manual definition of inputs.

method *m* to the list of target methods for the Input instance). Also, we rely on Soot to extract string literals to be used for testing (see Section 3.5).

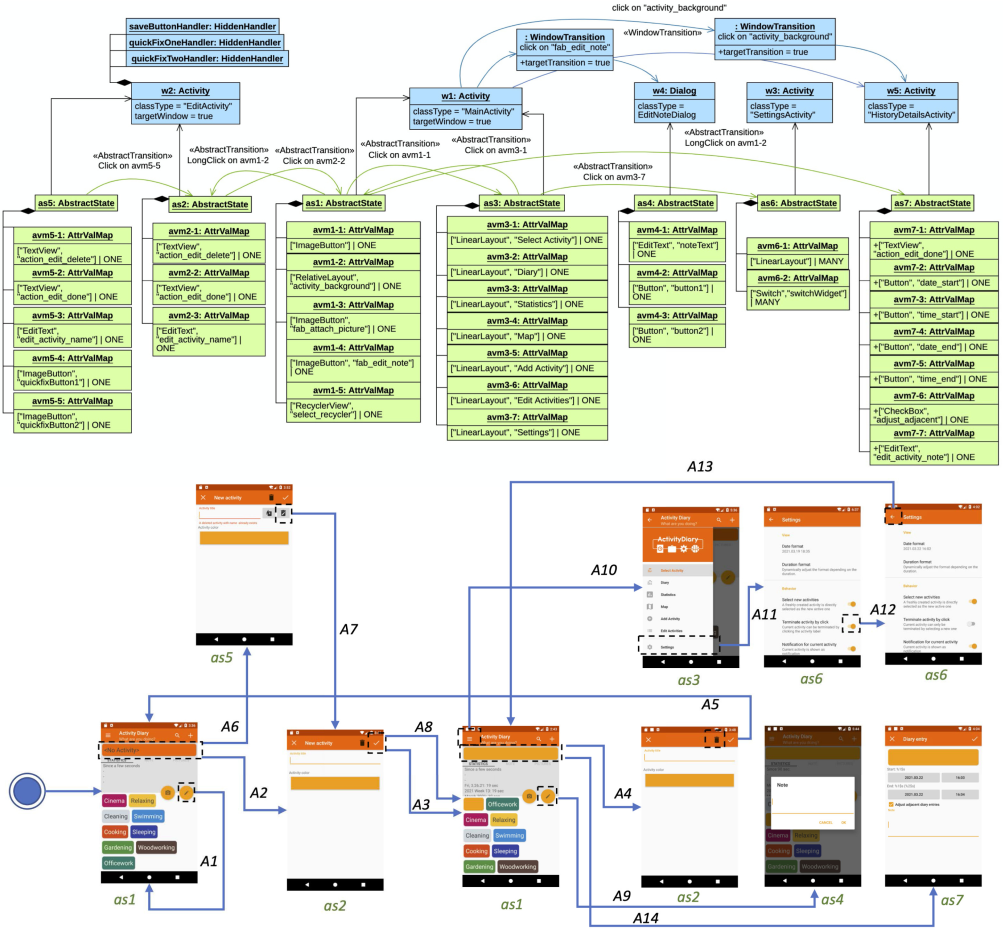
Finally, we determine if Gator does not identify some of the event handlers of the App, which is a common problem of static analysis tools for Apps. Indeed, these static analysis tools rely on hardcoded procedures for the identification of event handlers (e.g., they look for specific method names [56]); since OS APIs are under continuous evolution, it is unlikely that static analysis tools will ever be able to identify all the event handlers of an App. To address this problem, we introduced into ATUA three solutions, one based on static analysis (described in the next paragraph) and two based on dynamic program analysis (described in Sections 3.5.2 and 3.5.4).

To identify missing event handlers using static analysis, we rely on the observation that if an event handler is not detected by Gator, then the backward traversal of the call graph performed by ATUA will not reach any event handler but will terminate in a method that (i) belongs to a Window class and (ii) is not invoked by any other method of the App under test. Such methods are likely event handlers invoked at runtime by the Android APIs. We refer to them as *hidden-handlers*. We keep track of all the hidden-handlers encountered during the analysis along with the list of updated methods reachable from them. We rely on this information during Step 5.

3.4 Step 4. Prepare Manual Inputs

Certain input values are unlikely to be automatically generated, consequently certain Apps' features might not be automatically exercised without an appropriate solution to handle such cases. In related work, these inputs are referred to as *Unlocking GUI Input Event Sequences* (hereafter, *unlocking inputs*, for brevity) [2]. Examples include login credentials, files of a specific type, and data to be received by the App under test through the Android Intent mechanism. To handle these cases, in ATUA, engineers specify unlocking inputs in a JSON file capturing a set of input values and the patterns identifying the Windows requiring such input values. An example of our input definition format is provided in Figure 7.

According to our format, engineers can specify one or more input insertion patterns (e.g., *BookInsertionAndSearch* in Figure 7, Line 2). For each pattern, they specify the Windows in which the pattern should be used (Line 3) and the widgets that should be used to provide the input data


click on "activity_background"
saveButtonHandler: HiddenHandler
quickFixOneHandler: HiddenHandler
quickFixTwoHandler: HiddenHandler
: WindowTransition
click on "fab_edit_note"
+targetTransition = true
«WindowTransition»
: WindowTransition
click on "activity_background"
+targetTransition = true
w2: Activity
classType = "EditActivity"
targetWindow = true
w1: Activity
classType = "MainActivity"
targetWindow = true
w4: Dialog
classType = EditNoteDialog
w3: Activity
classType = "SettingsActivity"
w5: Activity
classType = "HistoryDetailsActivity"
«AbstractTransition»
Click on avm5-5
«AbstractTransition»
LongClick on avm1-2
«AbstractTransition»
Click on avm2-2
«AbstractTransition»
Click on avm1-1
«AbstractTransition»
Click on avm3-1
«AbstractTransition»
LongClick on avm1-2
«AbstractTransition»
Click on avm3-7
as5: AbstractState
as2: AbstractState
as1: AbstractState
as3: AbstractState
as4: AbstractState
as6: AbstractState
as7: AbstractState
avm5-1: AttrValMap
["TextView", "action_edit_delete"] | ONE
avm5-2: AttrValMap
["TextView", "action_edit_done"] | ONE
avm5-3: AttrValMap
["EditText", "edit_activity_name"] | ONE
avm5-4: AttrValMap
["ImageButton", "quickfixButton1"] | ONE
avm5-5: AttrValMap
["ImageButton", "quickfixButton2"] | ONE
avm2-1: AttrValMap
["TextView", "action_edit_delete"] | ONE
avm2-2: AttrValMap
["TextView", "action_edit_done"] | ONE
avm2-3: AttrValMap
["EditText", "edit_activity_name"] | ONE
avm1-1: AttrValMap
["ImageButton"] | ONE
avm1-2: AttrValMap
["RelativeLayout", "activity_background"] | ONE
avm1-3: AttrValMap
["ImageButton", "fab_attach_picture"] | ONE
avm1-4: AttrValMap
["ImageButton", "fab_edit_note"] | ONE
avm1-5: AttrValMap
["RecyclerView", "select_recycler"] | ONE
avm3-1: AttrValMap
["LinearLayout", "Select Activity"] | ONE
avm3-2: AttrValMap
["LinearLayout", "Diary"] | ONE
avm3-3: AttrValMap
["LinearLayout", "Statistics"] | ONE
avm3-4: AttrValMap
["LinearLayout", "Map"] | ONE
avm3-5: AttrValMap
["LinearLayout", "Add Activity"] | ONE
avm3-6: AttrValMap
["LinearLayout", "Edit Activities"] | ONE
avm3-7: AttrValMap
["LinearLayout", "Settings"] | ONE
avm4-1: AttrValMap
["EditText", "noteText"] | ONE
avm4-2: AttrValMap
["Button", "button1"] | ONE
avm4-3: AttrValMap
["Button", "button2"] | ONE
avm6-1: AttrValMap
["LinearLayout"] | MANY
avm6-2: AttrValMap
["Switch","switchWidget"] | MANY
avm7-1: AttrValMap
+["TextView", "action_edit_done"] | ONE
avm7-2: AttrValMap
+["Button", "date_start"] | ONE
avm7-3: AttrValMap
+["Button", "time_start"] | ONE
avm7-4: AttrValMap
+["Button", "date_end"] | ONE
avm7-5: AttrValMap
+["Button", "time_end"] | ONE
avm7-6: AttrValMap
+["CheckBox", "adjust_adjacent"] | ONE
avm7-7: AttrValMap
+["EditText", "edit_activity_note"] | ONE
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
as5
as1
as2
as1
as2
as4
as7
as3
as6
as6

Legend: The top part shows the EWTG, the DSTG is in the middle, while, to simplify reading, the GSTG is represented by means of screenshots corresponding to each GUITree. Labels below screenshots are used to associate GSTG states to AbstractStates. The Actions appearing in the GSTG are: *A1*, click on *EditNote* button. *A2*, long click on *Current activity* widget. *A3*, click on *Save* button. *A4*, long click on *Current activity* widget. *A5*, click on *Delete activity* button. *A6*, long click on *Current activity* widget. *A7*, click on the button to automatically rename the deleted activity. *A8*, click on *Save* button. *A9*, click on *EditNote* button. *A10*, click on *Open navigation* button. *A11*, click on *Settings* button. *A12*, click on *Terminate activity by click* button. *A13*, click on *Back* button. *A14*, click on *Current activity* widget.

Fig. 8. App Model for the Activity Diary running example.

specified (i.e., *DataFields* field in Line 4). Each widget is identified by a name (e.g., *isbn* in Line 5) and a regular expression that enables its selection in the GUITree, based on its name (e.g., *isbn_txt* in Line 6). Finally, multiple input instances (e.g., book names, in this case) can be specified (see field *Instances* in line 13).

Since ATUA relies on software engineers to identify unlocking inputs, its effectiveness might be affected by engineers' mistakes. For example, in our experiments, we specify manual inputs only for login operations and key features on the App under test (see Section 4.2), thus potentially omitting unlocking inputs concerning other App features. The integration of state-of-the-art solutions to automatically discover unlocking inputs is part of our future work [2].

3.5 Step 5. Automatically Test the App

ATUA automatically tests the updated App by triggering the Actions required to exercise target Transitions. When testing starts, the App model consists of an instance of the EWTG for the App under test. GSTG and DSTG are dynamically constructed and extended at runtime by ATUA.

The test execution process includes three distinct phases, each one relying on a different strategy for the generation of Actions. In *Phase 1*, ATUA triggers one Action for every target Input. The goal of Phase 1 is to handle the simplest scenario, i.e., exercise instructions that are executed every time data provided through a target Input is processed. In *Phase 2*, ATUA exercises target Windows with multiple, diverse sets of Inputs. The goal of Phase 2 is to exercise those instructions that are executed only when specific constraints on input values provided in a Window are satisfied. In *Phase 3*, ATUA exercises both target Windows and Windows they depend on. The goal of Phase 3 is to exercise those instructions that can be executed only when certain constraints on the input values provided in related Windows (e.g., preferences Windows) are satisfied.

3.5.1 Running Example. To illustrate our approach, we describe part of the actions taken by ATUA when testing the upgrade to version 134 of Activity Diary. Activity Diary enables end-users to record a diary for their activities. It includes features to categorize activities, report statistics, remind users about recurrent activities, and attach notes and pictures to activities.

We consider a subset of the features updated in version 134 of Activity Diary, which aim to (1) visualize the details of the current activity by clicking on the activity name, (2) edit the current activity or create a new activity with a long click on the current activity name, (3) automatically fix a duplicated name for an activity, (4) edit an activity note. Figure 8 shows the App model for our running example.

In the EWTG of Figure 8, the transition between MainActivity (*w1*) and EditNoteDialog (*w4*) is a target transition, since it triggers one modified method: the handler of the EditNote button (i.e., *editNoteHandler*, not shown in Figure 8). The EditActivity Window (*w2*) is a target Window, since it contains three hidden-handlers: *saveButtonHandler*, *quickFixHandlerOne*, and *quickFixHandlerTwo*. The hidden-handler *saveButtonHandler* reaches the modified method *checkConstraints,* while the other two hidden-handlers are the event handlers for the quick fix buttons appearing in the UI. ATUA classifies these three methods (i.e., *saveButtonHandler*, *quickFixHandlerOne*, and *quickFixHandlerTwo*) as hidden-handlers, because they are not associated by GATOR to any WindowTransition in the EWTG (see Section 3.3). Indeed, GATOR cannot correctly process the control flow that reaches function *setListener*, which is the function used to assign the three handlers to their corresponding buttons.[4]

The GSTG in Figure 8 captures the sequence of Actions triggered by ATUA during testing.[5] They are described in the following sections, where appropriate.

3.5.2 Detection of the Active Window. A building block of our test automation strategy is the detection of the active Window. More precisely, ATUA should determine if a pop-up (i.e., a Dialog, an OptionsMenu, or a ContextMenu) is open on top of the active Window. Neither Android nor DM2 provides such information. To determine if a pop-up is open, ATUA relies on the dimensions of the active Window on the screen. Indeed, if an Activity is displayed, then its dimensions should match the dimensions of the screen. Otherwise, the currently displayed Window is either a Dialog, an OptionsMenu, or a ContextMenu. To determine which Dialog or Menu is open, ATUA identifies the Window of the EWTG with the highest portion of Widgets visualized on the screen. ATUA

[4] GATOR does not correctly process control flows starting within event handlers, likely because of their recursive nature; in our running example, the control flow starts within event handlers triggered by changes in color selectors and text boxes.

[5] A demo video for the running example is available online [45].

computes the ratio, R_w, of Widgets belonging to Window w that appear in the displayed GUITree. More precisely, ATUA computes R_w for all the Dialogs and Menus that can be triggered by the current Window. The active Window is the one with the highest values for R_w.

If, due to the limitations described in Section 3.3, static analysis does not detect that, for a certain Window w, there is an event handler that will pop-up a certain Dialog or Menu p, then ATUA may not be able to find a Dialog or Menu to be matched with the displayed GUITree. More precisely, ATUA may observe that (1) the current Window has no Dialogs and Menus associated to it in the EWTG or (2) the score computed for every Dialog and Menu of the current Window is zero. To overcome such a problem, in these scenarios, ATUA updates the EWTG to include a new pop-up Window.

3.5.3 ATUA Testing Algorithm. At runtime, after detecting the active Window, ATUA automatically derives the current AbstractState according to the procedure described in Section 3.1. The subsequent activities depend on the current testing phase.

The activities performed in the three phases follow the same algorithm, which is presented in Algorithm 1. What differentiates the three phases are the strategies adopted to exercise the App and the test budget allocated. Line 1 in Algorithm 1 shows that the algorithm iterates till the test budget for the current phase is consumed (function *phaseBudgetConsumed*), all the targets for the current phase have been covered (function *coverageTargetsExercised*), or it cannot further improve coverage (function *stagnation*).

The iteration starts by identifying a test target (function *selectTarget*, Line 3). The test target is either a Window or a WindowTransition. A new test target is identified when no target has been selected yet or the current target has already been fully exercised (Line 2). After identifying the test target, ATUA relies on the App model to identify the test target path (Line 4), i.e., a sequence of Actions that makes the App render the target Window or reach the target AbstractState.

The test target path is derived with a breadth-first traversal of the App model. The traversal starts from the current AbstractState. The traversal proceeds through both AbstractTransitions and WindowTransitions. A WindowTransition is taken only if an AbstractTransition is not available. The visit of the model stops when we reach the test target or all the reachable nodes are explored.

So long as a test target is not reached (Line 8), ATUA executes function *reachTargetNode* (Line 9), which triggers the next Action in the test target path. For each Action in the test target path, we know the Window or the AbstractState to expect. After executing an Action, function *reachTargetNode* checks if the App is in the expected Window or AbstractState. If not, then function *reachTargetNode* flags the target as not reached and returns to the main execution loop to look for a different path to reach the test target (Line 5). When a target is reached (Line 10), ATUA exercises the target according to the Action generation strategy for the current phase (function *exerciseTarget*).

Finally, random exploration of the active Window might be triggered by functions *reachTargetNode* and *identifyPathToTarget* to improve the EWTG (Line 14). This is described in Section 3.5.4.

To regulate the allocation of the phase budget (i.e., how many Actions each function invoked by the algorithm is allowed to generate), the ATUA algorithm makes use of three budget variables: (1) *reachabilityBudget*, which specifies the maximum number of Actions to be used to reach a target node, (2) *targetBudget*, which specifies the number of Actions to be used to exercise the target node, (3) *randomBudget*, which specifies the number of Actions to be used for random exploration. At runtime, when counting the number of Actions performed, we ignore Actions of type TextInput and Click on checkboxes, since they generally do not trigger WindowTransitions. The budget variables are initialized with different values, depending on the current test phase. Table 3 provides an overview of the criteria adopted, which are described in detail in the following paragraphs. To

ALGORITHM 1: ATUA testing algorithm.

```
while (NOT stagnation()) AND (NOT phaseBudgetConsumed(phaseBudget)) AND (NOT coverageTargetsExercised()) do
    if target not selected OR target already exercised OR visitBudget exhausted then
        selectTarget()
        identifyPathToTarget()
    else if target unreachable then
        identifyPathToTarget()
    end if
    if NOT targetReached() then
        reachTargetNode(reachabilityBudget)
    else
        exerciseTarget(targetBudget)
    end if
    if additional random exploration required then
        performRandomExploration(randomExplorationBudget)
    end if
end while
```

Table 3. Strategies Adopted, in Different ATUA Phases, to Define the Budget Allocated to Distinct Test Activities

Phase	Budget	Strategy
Phase1	Phase	Infinite, i.e., all the target windows are exercised till stagnation is detected or all the targets are covered.
	Reachability	Infinite, i.e., all the paths are traversed in this phase.
	Target	Infinite, i.e., all the target Inputs are tried in this phase.
	Random Exploration	Set to *scaleFactor* · *NumberOfActionsForActiveWindow*. *NumberOfActionsForActiveWindow* is the number of distinct Actions that can be performed in the active window; it is based on the interactive information associated to a widget (e.g., we perform a Click Action if the widget is clickable, or four Swipe Actions—one for each swipe direction—if it is scrollable). In our experiments, we set *scaleFactor* to 1 for an overall test budget of one hour, to 2 for a test budget of five hours. Random exploration is triggered by either *reachTargetNode* or *identifyPathToTarget*.
Phase2	Phase	Set to *scaleFactor* · *NumberOfTargetWindows*.
	Reachability	Set to *scaleFactor* · *actionsThreshold*. The value is reset every time a new TargetWindow is identified. We set *actionsThreshold* to 25.
	Target	Set to be equal to what remains of the ReachabilityBudget after the target window is reached. In other words, *ReachabilityBudget*+*TargetBudget* = *scaleFactor*∗*actionsThreshold*.
	Random Exploration	Set to *scaleFactor* · *randomThreshold*. Random exploration is triggered by either *reachTargetNode* or *identifyPathToTarget*. We set *randomThreshold* equal to *actionsThreshold*.
Phase3	Reachability	Not used in this phase.
	Target	Set to *scaleFactor* · *actionsThreshold*. It is reset every time a new TargetEvent is identified.
	Random Exploration	Set to *scaleFactor* · *actionsThreshold*. Random exploration is triggered (1) to explore the related Window, (2) when the related Window cannot be reached through the identified path, (3) when the target Window cannot be reached through the identified path (see Section 3.5.7). We set *randomThreshold* to 5.

define budgets, a *scale factor* is used to optimally distribute the test budget across phases and test targets. For example, with a test budget of five hours, we can invest more time in Phase 2 than with a test budget of one hour.

3.5.4 Random Exploration. Functions *reachTargetNode* and *identifyPathToTarget* in Algorithm 1 may trigger the random exploration of the App under test to improve the EWTG. This is done when Inputs cannot be exercised and a test target cannot be reached.

Function *reachTargetNode* determines that an Input cannot be exercised when the associated Widget is not visible or enabled in the GUITree. It happens, for example, when the content of a *NavigationDrawer* varies based on the buttons pressed in the active Window. To make the required

Table 4. ATUA Input Generation Procedures

Widget type	Input generation procedure
Any widget	Trigger an InputEvent among the ones for which an event handler has been declared.
Textarea	Randomly apply one of the following: (1) leave it empty, (2) reuse a string already used in the past, (3) reuse a string already used for the same widget, (4) reuse a string literal extracted with static analysis, (5) use a randomly generated alphabetic string [12], (6) use a randomly generated non-alphabetic char.
Radio buttons and check boxes	Randomly select one of the possible options (e.g., checked/not checked, for check boxes).
Widgets with manual input	Randomly select one of the available InputInstances, if more than one is available, and then assign the specified value.
Intent	If the current activity declares an Intent, then it triggers the Intent specified by the engineer.

Widget visible, function *reachTargetNode* randomly exercises the active Window. This is done by iteratively and randomly selecting one widget among the ones that have been exercised less frequently in the active Window. The selected widget is then exercised according to the strategies listed in Table 4. The exploration of the active Window terminates when the desired widget is found or when the test budget for random exploration is exhausted.

Function *identifyPathToTarget* may determine that it is not possible to find a path to a test target. This happens when the EWTG does not include all the WindowTransitions, which is due to the limitations of static analysis tools mentioned in Section 3.3. For example, Gator does not detect the Animation design pattern [31], which leads to a WindowTransition. When a test target path is not found, ATUA performs a random exploration of the active Window and then resumes the execution from the beginning of the main execution loop. ATUA records of all the unreachable targets identified.

3.5.5 Phase 1. In Phase 1, a test target is any Window with target Inputs that have not been triggered yet. Function *selectTarget* randomly selects a Window with such characteristics (Line 3 in Algorithm 1).

After selecting the target Window, ATUA follows the path to reach the test target.

Function *exerciseTarget*, first produces *user-like inputs* (i.e., input values for text areas, radio buttons, and check boxes), as specified in Table 4. Then it triggers an Action that exercises a randomly selected target Input. Function *exerciseTarget* keeps triggering Actions that exercise target Inputs until all the target Inputs have been exercised or another Window has been visualized. ATUA then resumes the execution of the main loop (i.e., Line 1 in Algorithm 1). When the target Window is associated to hidden-handlers that can reach target methods, ATUA also performs a random exploration of the Window. If an Action triggers the execution of an hidden-handler, then ATUA introduces a corresponding WindowTransition into the EWTG.

To maximize the chances of exercising every target Input, which is the objective of Phase 1, the phase, reachability, and target budgets are infinite. More precisely, we try to reach every TargetWindow (infinite *phaseBudget*) by trying every possible path (infinite *reachabilityBudget*); furthermore, we exercise every TargetWindow with all the target Inputs (infinite *targetBudget*).

In Phase 1, we observe *stagnation* when all the remaining targets either cannot be reached or all their target Inputs cannot be exercised.

Running Example. In Phase 1, ATUA triggers one Action for every target input. By default, Activity Diary starts by rendering the MainActivity with a predefined set of activities and no current activity being selected. Since MainActivity is a target Window, ATUA exercises it. First ATUA clicks on the edit note button (Action A1 in Figure 8) and, since there is no current activity selected, Action A1 partially covers the target methods (indeed, Activity Diary does not open the EditNote

Window, which will be achieved in Phase 2). Then, ATUA triggers a long click on the current activity widget (A2), which leads to an instance of the EditActivity Window. Since EditActivity is a target Window, ATUA aims to exercise it. However, EditActivity contains only hidden-handlers, not target Transitions. For this reason, ATUA performs a random exploration of the window; during the random exploration, after filling the window with random inputs, ATUA clicks on the save button (A3), which triggers the execution of one of the updated methods (i.e., *checkConstraints*) and thus ATUA introduces into the DSTG a new AbstractTransition associated to the updated method (i.e., the abstract transition between *avm2-2* and *as1* in Figure 8, which does not have a corresponding WindowTransition in the EWTG). After these three actions, ATUA has exercised both the MainActivity and the EditActivity and can thus move to Phase 2. In Phase 1, ATUA has thus exercised all the easy-to-reach target methods in a few steps.

3.5.6 Phase 2. In Phase 2, we aim to maximize the coverage of those target methods that have not been fully covered. To this end, the target Window shall be the one that can trigger the execution of the highest number of uncovered instructions. Also, since the AbstractState of an App might affect the reachability of a target method, we should give higher priority to Windows with target methods exercised in fewer AbstractTransitions.

To achieve the above-mentioned objectives, we select as target Window the one that maximizes the score WS_w,

$$WS_w = \sum_{m \epsilon MT_w} c_w \cdot u_m,$$

where MT_w is the set of target methods associated to the target Inputs of Window w. Term u_m is the number of uncovered instructions belonging to method m. A target Window x can thus be any Window with $WS_w > 0$. A Window w is selected as target with a probability proportional to WS_w.

In the formula above, c_w is a weight introduced to focus first on those methods that have been covered less. It is the complement of the proportion of AbstractTransitions that exercise the method:

$$c_m = 1 - \frac{AA_m}{AA},$$

where AA_m is the number of AbstractTransitions that covered method m and AA is the number of AbstractTransitions in the App model.

To select the test target path, ATUA identifies the AbstractState with the highest number of uncovered instructions belonging to interactive widgets, which is captured by the AS_{as_w} score:

$$AS_{as_w} = \sum_{m \epsilon MT_{as_w}} c_w \cdot u_m,$$

where as_w is an AbstractState for the Window w, and MT_{as_w} is the set of target methods that might be covered through as_w. The set MT_{as_w} consists of all target methods triggered by either (1) an Intent or (2) an InputEvent for an interactive widget in as_w.

Function *exerciseTarget* works in the same way as in Phase 1. However, Phase 2 differs from Phase 1 with respect to the target, phase, and reachability budgets. Indeed, to uniformly distribute the phase budget across the selected target Windows, in Phase 2, the budget for reaching a test target and exercising it is set to "*scaleFactor · actionsThreshold*," with *actionsThreshold* representing a number of Actions that, based on preliminary experiments, is sufficient to reach a target and exercise it (in our experiments, we set its value to 25). In Phase 2, the *phase budget* is exhausted when ATUA has exercised a number of windows that is equal to "*scaleFactor· overall number of TargetWindows.*" Also, a same Window can be selected as a target multiple times. By repeatedly generating different sets of Actions for a same Window, ATUA covers different combinations of user-like inputs, which may include combinations that lead to the coverage of different sets of instructions.

In Phase 2, we observe *stagnation* when, after exercising all the available targets, the coverage of target methods has not increased.

Running Example. Phase 2 is necessary to maximize the coverage of updated methods reached through the MainActivity and the EditActivity Windows. The target Window with the highest WS_w score is EditActivity, because some of the instructions of method *checkConstraints* and all the instructions implementing the quick fix feature have not been exercised in Phase 1. MainActivity has a lower WS_w score, since only a few instructions of the EditNote button handler are not covered.

ATUA selects the EditActivity Window as first target; at this stage, it has only one AbstractState (i.e., *as2* in Figure 8). ATUA reaches the EditActivity Window with a long click (Action A4) on the current activity (an activity with an empty name). EditActivity includes one target AbstractTransition, the one exercising *checkConstraints*. While generating inputs to maximize the coverage of method *checkConstraints*, ATUA clicks on the button that deletes the activity and brings the App back to the MainActivity (A5). From the main Activity, ATUA performs again a long click on the currentActivity widget (A6), which leads to an instance of EditActivity for the definition of a new activity where the quick fix buttons are visualized. In this case, the quick fix buttons are visualized, because an activity with an empty name (the default for new activities) had been selected and Activity Diary already contains an activity with an empty name (i.e., the one deleted by Action A5). Because of the presence of the two quick fix buttons, ATUA introduces a new abstract state into the DSTG (i.e., *as5*).

To cover the hidden-handlers of EditActivity, ATUA exercises the quick fix button that automatically renames the activity having a conflicting name (A7). Finally, ATUA selects MainActivity as target and exercises the EditNote button (A9), which pops up the EditNote Dialog, thus covering the missing lines. In Phase 2, ATUA successfully covered all the target methods triggered within a target Window (i.e., EditActivity) by repeatedly exercising it.

3.5.7 *Phase 3.*

In Phase 3, we aim to cover those target method instructions that exhibit data dependencies from state variables defined by Windows different than the one reaching a target method. Examples include instructions that can be executed only after enabling specific options in the preferences Window of the App. For this reason, in Phase 3, the test target is a WindowTransition presenting associated targetMethods that remain to be fully covered. Also, for each WindowTransition to be tested, we need to identify a set of related Windows that should be exercised before executing it.

Function *selectTarget* returns a WindowTransition belonging to a target Window selected according to the same criteria as for Phase 2, i.e., with a probability proportional to its *WS* score. To minimize the effort spent in reaching target Windows, once a target Window has been selected, function *selectTarget* iteratively returns each target WindowTransition belonging to it.

When a test target has been selected, in function *exerciseTarget*, ATUA (1) identifies the related Window that should be exercised first, (2) identifies a path to this related Window, (3) reaches the related Window and randomly exercises it, (4) identifies a path to the closer AbstractState for the target Window in which the target WindowTransition is enabled, and (5) reaches the identified AbstractState and triggers a target Input. In Phase 3, function *identifyPathToTarget* is not invoked, because testing starts from the related Window. Consistent with Phase 2, the *targetBudget* is set to $scaleFactor \cdot actionsThreshold$.

Function *exerciseTarget* relies on random exploration (1) to explore the related Window, (2) when the related Window cannot be reached through the identified path, (3) when the target Window cannot be reached through the identified path. The random exploration budget is set to *scaleFactor* · *randomThreshold*. Since random exploration has been largely used in previous phases and to limit the time spent in related windows, in Phase 3, we set *randomThreshold* to a value lower than the one used in Phase 2 (e.g., we used five in our experiments).

To identify related Windows, we rely on information retrieval techniques. We do not rely on traditional data-flow analysis [8], because data dependencies might be implemented in many different forms (e.g., setting a state variable in a shared object or saving a property in a key-value registry) that are not fully identified by such analysis.

Related Windows are retrieved through the computation of the **term frequency (TF)** and **inverse document frequency (IDF)** metrics, which are standard information retrieval metrics [42]. In the following, we discuss how we compute these metrics.

Since dependencies between Windows are due to either state variables defined in shared objects or property values in key-value registries, the executable code of Windows presenting such dependencies should share a subset of *class attributes* and *literals*. For this reason, the terms used to identify dependencies are *class attributes* and *literals* appearing in the implementation of the methods of the App (extracted with Soot).

We compute $TF(t, h)$, the frequency of the term t for an Input handler h, as the number of methods in which the term appears, considering the handler itself and any of the methods invoked by the handler. To include only terms that characterize the functionality triggered by the WindowTransition, we consider only methods declared in the same class of the handler or in its inner or outer classes.

The frequency of the term t for a WindowTransition wt is computed as the sum of the term frequency for all the handlers of the Input (HI_{wt}) that triggers the transition,

$$TF(t, wt) = \sum^{h \in HI_{wt}} TF(t, h).$$

The frequency of the term t for a Window w, $TF(t, w)$, instead, is computed as the number of methods in which the term appears, considering the methods that are either declared in the class that implements the Window or in its parent class.

The inverse document frequency of a term is computed as

$$IDF(t) = log\left(\frac{total\ number\ of\ Windows}{number\ of\ Windows\ in\ which\ t\ appears}\right).$$

The related Windows for a WindowTransition wt can be identified by computing a dependency score for every Window w of the App, as follows:

$$DS(w, wt) = \sum^{t \in T} NW(t, w) \cdot NW(t, wt),$$

where T is the set of terms for the App, and $NW(t, d)$ is the normalized term weight, which captures the extent to which a term is representative for either a Window or a WindowTransition. $NW(t, d)$ is computed according to a standard formula [42]:

$$NW(t, d) = \frac{TF(t, d) \cdot IDF(t)}{EL(d)},$$

where EL(d) is the Euclidean Length of the element d (i.e., a Window or a WindowTransition). It is computed as the square root of the sum of the terms' weight squared [42].

ATUA randomly selects related Windows using the dependency score as probability distribution.

Phase 3 terminates when the overall test budget is exhausted or all the instructions of the target methods have been covered.

Running Example. Phase 3 enables ATUA to test the Activity Diary feature that visualizes the details of the current activity after a click on the activity name. This feature requires a specific configuration to be enabled in the settings page (by default, a click on the activity name terminates the activity). After selecting the MainActivity as target Window (EditActivity had been fully exercised in Phase 2), ATUA selects the SettingsActivity as related Window to be exercised first (it is reached with Actions A10 and A11 in Figure 8). While exercising the SettingsActivity, it deselects the option *Terminate activity by click* (Action A12). After exercising the related Window, ATUA reaches the MainActivity (A13) and triggers the Action that exercises the modified feature (i.e., click on current activity widget—A14). Phase 3 thus enabled ATUA to test the updated feature (i.e., visualize the current activity's details after a click) in a few steps, which is unlikely with random-based, state-of-the-art approaches (see Section 4.5 for additional examples).

4 EMPIRICAL EVALUATION

The objective of the empirical evaluation is to compare ATUA with state-of-the-art approaches in terms of cost-effectiveness. It is motivated by our need to achieve high test coverage (effectiveness) while enabling the verification of test results within an acceptable budget (cost) in a CI context.

When a new release is ready for testing, a test automation technique is *effective* when it enables engineers to verify updated features in the App under test automatically; more precisely, when it extensively exercises updated methods and their instructions. Measuring the effectiveness of App testing techniques in terms of method and instruction coverage is common practice [16]. Although engineers may aim to exercise all the methods that could be impacted by the changes (e.g., the ones identified by means of change impact analysis as mentioned in Section 3), in our empirical evaluation, we focus on updated methods, since exercising them is the minimum requirement of any testing criterion targeting software updates.

What we refer to as *cost* comes in two forms, (1) test execution time and (2) human effort , which we define as the time spent by a trained engineer to execute the manual tasks required by a test automation technique. In general, *test execution time* does not necessarily need to be minimized, but it should be practical. For example, we expect a test budget of one hour to be practical in a continuous integration context, while a budget of five hours might be acceptable when testing overnight.

Human effort, in our context, is mostly driven from the absence of a solution to automate test oracles, even in the presence of automated test input generation. More precisely, it is not possible to define automated oracles working with any feasible test input; consequently, the evaluation of the correctness of App outputs should, in general, be done manually. As described in Section 3, to address the oracle problem in the context of App updates, engineers can rely on two complementary state-of-the-art approaches that, respectively, address regression failures and failures in newly implemented, repaired, or modified features. To discover regression failures, engineers can replicate, on a previous App version, the test input sequences generated for the updated App. With ATUA, input sequences correspond to the sequences of Actions generated by ATUA to test an App. In such context, they address the oracle definition problem by verifying that the outputs generated by the two App versions match. To discover failures in new, repaired, and modified features engineers should visualize the screenshots of the Windows or the GUITrees generated for the provided inputs. The effort in doing so can potentially be reduced through crowdsourcing. In this context, the oracle definition problem remains unaddressed; a manual oracle can be defined (i.e., a human can decide if an output is correct based on the App specifications) but oracle evaluation remains manual and, therefore, expensive. Regardless of the situation and context, human effort, given the scarcity of qualified human resources, should in general be minimized.

We assume that human effort is proportional to the number of inputs generated by test automation. Indeed, to overcome execution errors due to changes in the GUI and be able to replicate test input sequences in a different App version, engineers may need to manually repair the sequence of inputs (e.g., by changing the ID of a widget to be clicked). A large number of inputs may thus lead to numerous repair operations and make test oracle automation infeasible.[6] Also, the number of outputs that shall be visually inspected by an engineer is proportional to the number of inputs; indeed, an engineer shall visualize the GUITree or the screenshots of the active Window rendered after each Action (see Section 3). The effort required to inspect a Window or a GUITree depends on the specific output generated by the App and the specification of the App. For example, in our running example (Figure 8), it is reasonable to believe that it is simpler to inspect the Window rendered after Action A4, which contains only a text box and a color selector, rather than inspecting the output of Action A10, which leads to a menu dialog with many textual items. However, the specifications of the App may also impact the required human effort. For example, the color visualized when editing an activity (i.e., A4) depends on the color previously selected for the same activity, which requires the engineer to verify such consistency based on execution history. The menu dialog visualized with Action 10, instead, does not depend on execution history and thus may require less human effort for verification. For these reasons, the cost for the visual inspection of results can be estimated only through an experiment involving human subjects. However, since our objective is to *compare the human effort required by different test generation techniques*, not to estimate the cost of failure detection approaches, we can rely on the number of inputs. Indeed, under the assumption that the distribution of the effort for visually inspecting an output is similar for different test automation techniques, we can conclude that techniques leading to a larger number of inputs may lead to proportionally increased costs for output verification. For our experiments, such assumption holds because all the considered approaches exercise a common subset of target methods and we expect a same method to lead to similar outputs across different executions. Indeed, we have observed that more than 50% of the target methods exercised in our experiments are covered by all the testing techniques and more than 80% of them are covered by at least two techniques (see Section 4.5).

Our research questions are organized according to the two cost measures above. They evaluate the extent to which we have achieved the objectives mentioned in the introduction; RQ1 addresses O2, while RQ2 addresses O1. Further, beyond comparisons, RQ3 aims to characterize how ATUA and state-of-the-art approaches complement each other.

RQ1. *Can ATUA reduce the human effort required for testing Apps, compared to state-of-the-art approaches?* We aim to determine if the number of inputs generated by ATUA is significantly lower than the number of inputs generated by state-of-the-art approaches, for a same execution time budget. A lower number of inputs makes test automation more widely applicable in practice, since it reduces the effort related to the definition of test oracles and repair of input sequences. Also, to determine if the effort of using ATUA is justified by practical benefits, we aim to verify if ATUA provides higher effectiveness per unit of effort than state-of-the-art approaches.

RQ2. *Can ATUA effectively test Apps within practical time budgets, compared to state-of-the-art approaches?* This research question aims to determine if ATUA performs significantly better

[6] Related work reports that repairing a single test input takes 15 minutes, on average [30].

than state-of-the-art approaches in terms of coverage of updated methods and their instructions, for a same execution time budget.

RQ3. *Is there any difference in the functionalities that are automatically exercised across test automation approaches?* We aim to determine if the testing approaches considered in our empirical evaluation are complementary and to what extent. Specifically, we aim to determine if there are differences in the inputs triggered by the different approaches (e.g., input sequence length, widgets being exercised, or program states being reached) that lead to a diverse and complementary set of functionalities being exercised.

A replicability package is made available online [45].

4.1 Study Subjects

To perform our experiments, we considered as experimental subjects a number of Apps available on the Android Play Store that are highly popular (i.e., more than 100,000 downloads, on average) and that were used for validation in recent papers reporting on related techniques, i.e., APE [29] and DM2 [12]. We considered only the Apps that can be executed on the recent Android simulator version supported by our toolset (i.e., Android API Level above 23). For each App, we considered the latest 10 released versions at the time of running our experiments (hereafter referred to as V0, . . . , V9), when available.[7] In Table 5, for each version of each subject, we report the overall number of methods, the number of updated methods, the overall number of bytecode instructions, and the number of bytecode instructions belonging to the updated methods. In total, we downloaded and processed 83 App versions, 74 being updated versions.

For all subjects, we treat version V0 as the first released version. The number of updated methods ranges from one (e.g., version V8 of subject App 2) to 1,430 (e.g., version V6 of subject App 1), thus being representative of a wide variety of release scenarios (i.e., from simple bug fixes to major releases). The number of bytecode instructions, ranging from 3,667 to 165,747, shows that the considered Apps have varied degrees of complexity. Further, the growing number of instructions belonging to the different versions of the Apps (e.g., from 32,208 to 69,856 in the case of Wikipedia) suggests that they are representative of typical Apps where features are incrementally introduced at every release, thus further motivating the adoption of ATUA.

4.2 Experimental Setup

In our experiments, we compare ATUA with three state-of-the art test automation tools, i.e., DM2, Monkey, and APE. These tools do not specifically target updated methods, but simply aim to maximize the coverage of the whole App.

DM2 has been selected, because it is the framework on top of which we implemented ATUA. We configured it to use the biased-random testing strategy, which matches the random input selection strategy of ATUA. The comparison with DM2 enables us to determine if the additional analyses performed by ATUA (i.e., static program analysis, adaptable state abstraction function, and inputs generation based on information retrieval) contribute to generating better results than a simpler solution based on dynamic program analysis only.

Monkey is a program that runs on the Android emulator and generates pseudo-random streams of events. It is used as baseline for App testing approaches and surprisingly fares better in many benchmarks [70]. The reason is that the time saved by not processing the App GUITree can be used to further explore the App state space.

[7] Preliminary experiments to set up ATUA had been conducted with Jamendo, a music streaming App [35].

Table 5. Overview of Subject Systems

	Subject App	Details: (V0)		V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
1	Wikipedia	**V**	110	144	146	159	190	198	10239	10263	10264	10269	
		AM	3,767	5,009	5,646	6,435	6,943	7,477	8,814	8,751	8,759	8,793	
		UM	3,767	446	195	108	370	292	1,430	535	13	94	
		AI	32,208	38,913	43,753	48,761	51,147	54,759	68,207	69,471	69,533	69,856	
		UI	32,208	11,000	4,157	2,441	6,606	6,345	24,536	12,724	281	2,698	
2	Activity Diary	**V**	105	111	115	117*	118	122	125	130	131	134	
		AM	260	333	333	333	333	450	479	540	540	659	
		UM	260	18	3	7	12	117	39	28	1	49	
		AI	3,667	4,832	4,831	4,834	4,880	6,613	7,052	8,247	8,251	10,622	
		UI	3,667	558	21	295	599	3,393	2,027	1,535	15	2,459	
3	File Manager	**V**	44	53	77	79	82	84					
		AM	2,042	2,132	3,422	3,430	3,648	3,648					
		UM	2,042	306	415	11	644	2					
		AI	34,389	34,931	48,241	48,294	51,755	51,789					
		UI	34,389	14,510	13,744	703	24,960	143					
4	Nuzzel	**V**	302	303	318*	323	325	328	329	330	331	333	334
		AM	4,223	4,220	4,524	4,498	4,527	4,650	4,771	4,832	4,833	4,833	4,834
		UM	4,223	8	717	75	33	41	21	21	1	1	1
		AI	40,522	40,449	43,309	43,083	43,403	44,234	45,331	45,908	45,913	45,916	45,940
		UI	40,522	151	18,335	2,952	1,593	1,990	647	1,378	35	69	45
5	Yahoo weather	**V**	1.16.0	1.16.1	1.16.2	1.17.3	1.18.1	1.19.1	1.20.1	1.20.3	1.20.5	1.20.7	
		AM	2,932	2,904	2,904	2,630	3,105	3,109	3,178	3,255	3,255	3,303	
		UM	2,932	5	4	243	10	16	118	101	12	9	
		AI	38,015	37,867	37,857	34,220	38,219	38,211	39,086	39,439	39,439	39,462	
		UI	38,015	417	272	10,198	857	588	4,295	3,842	689	961	
6	Wikihow	**V**	2.7.3	2.8.0	2.8.1	2.8.3	2.9.1	2.9.2	2.9.3				
		AM	333	333	333	333	325	322	319				
		UM	333	111	1	1	65	4	18				
		AI	3,704	3,992	3,941	3,944	3,808	3,761	3,657				
		UI	3,704	2,279	39	42	1,370	93	543				
7	BBC Mobile	**V**	5.1.0	5.10.0	5.11.0	5.12.0	5.13.0	5.4.0	5.5.0	5.6.0	5.8.1	5.9.0	
		AM	10,706	8,724	8,792	8,902	8,926	9,945	10,380	10,696	10,200	8,939	
		UM	10,706	649	27	44	25	603	242	553	77	95	
		AI	76,649	61,604	62,078	62,937	63,232	71,053	73,082	73,950	72,439	61,618	
		UI	76,649	11,182	1,557	2,288	2,101	10,637	6,324	9,484	1,638	3,274	
8	VLC player	**V**	3.1.4	3.1.5	3.1.7	3.2.12	3.2.2	3.2.3	3.2.6	3.2.7	3.2.9		
		AM	6,796	6,843	6,854	8,681	8,544	8,551	8,621	8,641	8,676		
		UM	6,796	672	26	3	149	13	51	33	42		
		AI	86,266	87,560	87,886	117,207	115,344	115,412	116,409	116,647	117,071		
		UI	86,266	34,086	1,522	150	9,611	1,163	3,527	1,961	3,010		
9	City-mapper	**V**	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.8	9.9	10.0	
		AM	9,629	9,499	9,599	9,491	9,602	9,761	9,868	9,929	9,884	10,050	
		UM	9,629	51	37	55	73	119	76	73	12	69	
		AI	155,117	154,086	157,036	153,200	155,950	161,914	164,267	165,747	165,747	163,303	
		UI	155,117	2,726	2,286	2,075	3,160	6,262	2,756	2,340	1,372	3,775	

Legend: V: ID of the version under test. AM: number of methods implemented in V (All Methods). UM: number of Updated Methods in V. AI: number of instructions in V (All Instructions). UI: number of instructions belonging to updated methods. An asterisk (*) is used to indicate not tested versions.

APE is a state-of-the-art App testing toolset that overcomes existing approaches, thanks to an adaptable state abstraction function (see Section 5).

In our experiments, we considered two possible execution scenarios with, respectively, test budgets of one hour (a practical choice in a continuous integration context) and five hours (a reasonable choice for overnight execution).

To use ATUA, for three subjects, we specified a set of manual inputs necessary to exercise the primary features of the Apps (e.g., to login and use the App). Support for manual inputs is a necessary feature of test automation tools, because Apps often require domain-specific information that cannot be derived automatically (e.g., login data). Table 6 provides a summary of the manual inputs defined; for each, we provide a description and the number of windows in which the manual input

Table 6. Case Studies with Manual Inputs

Case study	Feature tested	# Windows	# Data fields	# Instances
Wikipedia	Log-in functionality	1	2	1
	Creation of a new account	1	4	14
VLC	Play a video stream using a URL	1	1	2
	Populate the library with all the videos on the device	1	1	1
Nuzzel	Request an e-mail newsletter	1	1	1

might be triggered. For Wikipedia, we configured two manual inputs, one with the information for creating a new account, another one with information to log-in. In the case of VLC, we provide the URL of a stream to be reproduced and the indication of a checkbox to be checked to populate the library with device data (otherwise, no content can be played and testing is limited). Regarding Nuzzel, we provide the email address to receive a newsletter. The effort required to define manual inputs is limited; indeed, for each input, we have specified a single Window where it is applied, between one and four data fields, and a very limited number of input instances, that is, one for every tested feature except for the creation of a new Wikipedia account and the playback of a video stream with VLC. When creating a new account, it is necessary to specify a larger set of inputs to exercise the feature under test multiple times; indeed, a same e-mail address cannot be shared by distinct Wikipedia accounts. As for VLC, since one of its main features is to play video streams, it makes sense to test it with both a working and a corrupted video stream. This example illustrates that the effort required to specify manual inputs is negligible.

To account for randomness, we executed each tool against each updated version 10 times. We report results for 72 of the 74 versions available, since, for two App versions of Nuzzel and Activity Diary (indicated with an asterisk in Table 5), it was not possible to execute all the testing tools. More precisely, for version 318 of Nuzzel, the App starts but gets stuck in the first Activity, while version 117 of ActivityDiary can be tested only with ATUA and DM2, but not with Monkey and APE. In total, we executed 5,760 test sessions (4 tools × 72 versions × 10 runs × 2 test budgets) for a total of 17,280 test execution hours. To perform our experiments, we relied on the Grid 5000 infrastructure [10, 32], which provides access to 800 compute-nodes grouped into homogeneous clusters. We rely on nodes with 16 × 2.1 GHz and 18 × 2.2 GHz CPU cores.

In the following sections, we analyze differences in results using a non-parametric Mann Whitney U-test (with $\alpha = 0.05$). Particularly, we discuss the p-values computed by the Mann Whitney U-test to reject null hypotheses stating that there is no difference between ATUA and each of the state-of-the-art solutions, a common practice in software testing research [6, 7]. We discuss effect size based on Vargha and Delaney's A_{12} statistics [68], a non-parametric effect size measure. The A_{12} statistic, given observations (e.g., code coverage, in our context) obtained with two treatments X and Y (testing tools, in our context), indicates the probability that treatment X leads to higher values than treatment Y. Based on A_{12}, effect size is considered small when $0.56 \leq A_{12} < 0.64$, medium when $0.64 \leq A_{12} < 0.71$, large when $A_{12} \geq 0.71$. Otherwise, the two populations are considered equivalent [68]. In contrast, when A_{12} is below 0.50, it is more likely that treatment X leads to lower values than treatment Y. Symmetrically to the case above, effect size is small when $0.36 < A_{12} \leq 0.44$, medium when $0.29 < A_{12} \leq 0.36$, large when $A_{12} \leq 0.29$.

4.3 RQ1: Human Effort

4.3.1 Experimental Setup. In line with the discussion concerning human effort reported above, to address RQ1, we count the number of inputs generated by each testing tool, for each test execution run. For DM2 and ATUA, we rely on the CSV file generated by the ActionTrace component of DM2, which reports all the inputs triggered during testing. For Monkey and APE, we record the number of test inputs reported by the tool at the end of execution.

Metrics. For each subject App, we compare *distributions of the number of inputs generated across tools*. We also analyze the *target instructions/input ratio*, that is, the ratio between the number of target instructions (i.e., instructions belonging to updated methods) that are automatically exercised and the number of inputs triggered by the test automation tool. This ratio captures how useful it is for a software engineer, on average, to invest time in repairing a single input of the test sequence or verifying the output produced by an input. For example, a target instructions/input ratio of five indicates that, for every input, the test automation approach exercises, on average, five instructions belonging to updated methods.

To answer positively this research question, ATUA, compared to other tools, should generate less test inputs and have the highest *target instructions/input ratio*.

4.3.2 Results. Figures 9 and 10 show boxplots capturing the number of inputs generated by each approach in every run, for every subject App, for the two distinct test budgets considered (i.e., one hour and five hours). Please note that, in all the boxplots presented in this article: (1) horizontal dashed lines show the average across the data points of the boxplot (i.e., average for the subject App), (2) horizontal dotted lines traversing the whole chart show the average across all the runs, (3) whiskers are used to report min and max values across runs for all versions.

Figures 9 and 10 show that ATUA generates, on average, the lowest number of inputs: 876.53 for one hour, 4,608.57 for five hours. ATUA is thus the most suitable approach to minimize test automation effort. Monkey generates the largest number of inputs (i.e., 57,888.79 for one hour, 291,614.69 for five hours), because it does not invest any of the time budget into analyzing execution data but simply generates purely random inputs. APE relies on Monkey to generate inputs; however, APE generates less inputs than Monkey (i.e., 27,505.89 for one hour, 134,640.71 for five hours), because it spends time refining the state abstraction function (see Section 5). Finally, DM2 generates a number of inputs (i.e., 1,325.71 for one hour, 6,858.16 for five hours) that is closer to those generated by ATUA. This is mostly due to the fact that both approaches are model-based and share the same dynamic analysis infrastructure; however, on average, ATUA generates less inputs, because it invests more of the time budget into the analysis of runtime data.

Figures 11 and 12 present the same boxplots as Figures 9 and 10 but zoom in on ATUA and DM2 data to highlight their differences. Table 7 reports the p-value and A_{12} statistics obtained with the Mann Whitney U-test and the Vargha and Delaney's method, respectively. Recall that we aim to minimize the number of inputs and are thus interested in effect sizes below 0.46, i.e., the probability that ATUA generates a number of inputs higher than another approach should be below 0.46, ideally close to zero.

For a budget of one hour, for all the subject Apps, ATUA generates, on average, less inputs than DM2. Differences are statistically significant (i.e., we reject the null hypothesis that *there is no difference in the number of inputs generated by ATUA and the state-of-the-art approach*) and effect size is always in favor of ATUA and large for seven of the nine subjects. Differences with Monkey and APE are always significant, and effect size is always large except for subject 5 (YahooWeather), in which APE does not interact properly with one App version, apparently because of a bug in APE.

With a budget of five hours, ATUA also generates, on average, less inputs than DM2 across subjects. But in the case of Wikipedia, effect size is not in favor of ATUA (i.e., it is likely to generate more inputs than DM2). However, this is due to a bug in DM2 rather than a feature; indeed, in the presence of WebViews, the communication between the DM2 device daemon and the DM2 client are delayed, thus reducing the number of inputs being generated. In the case of ATUA, which

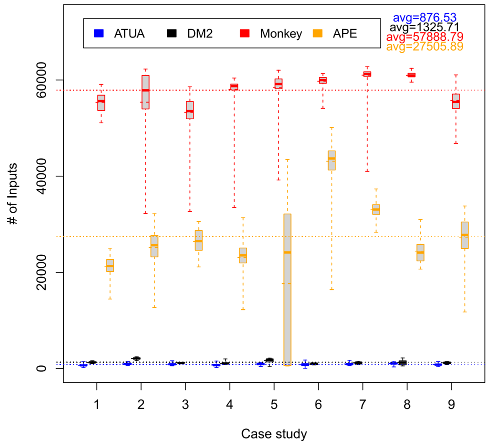


Fig. 9. Number of inputs generated (budget = 1 hour).

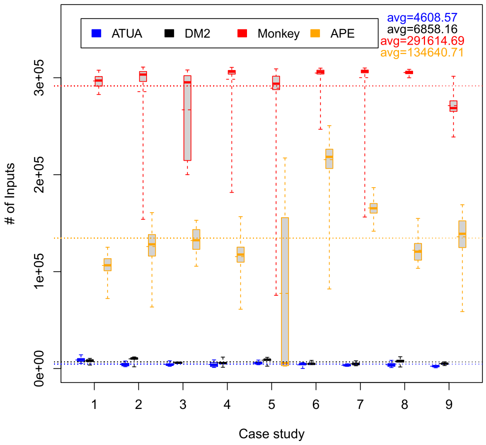


Fig. 10. Number of inputs generated (budget = 5 hours).

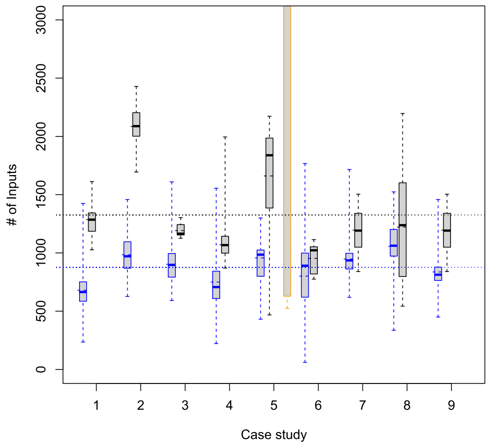


Fig. 11. Number of inputs generated (budget = 1 hour). Zoom on ATUA and DM2 data.

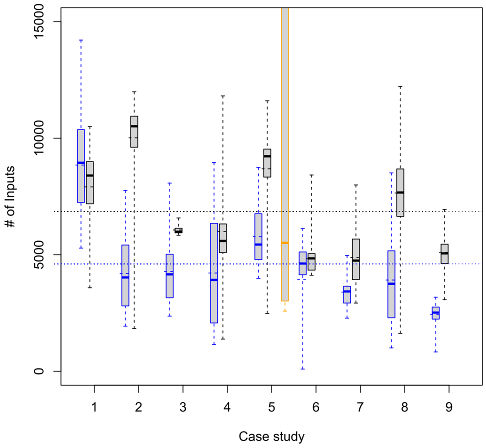


Fig. 12. Number of inputs generated (budget = 5 hours). Zoom on ATUA and DM2 data.

is built on top of DM2, this problem is less evident, because such communication is triggered less frequently. The differences between the number of inputs generated by ATUA and the ones generated by APE and Monkey are always statistically significant with a large effect size in favor of ATUA (except for YahooWeather, as already discussed above).

Figures 13 and 14 show, for each subject, the distribution of the target instructions/inputs ratio. ATUA has the highest ratio: 2.26 for one hour, 0.49 for five hours. For the one-hour budget, ATUA's test automation effort (i.e., manual repair of a GUI input, visual inspection of outputs) is thus more beneficial, because each input enables the verification of 2.26 additional target instructions. As a comparison, other state-of-the-art approaches yield lower ratios: 1.34 (DM2), 0.03 (Monkey), and 0.10 (APE). These results show that, though Monkey and APE are known for effectively triggering crashes, they are unlikely to be applicable in a testing context where the number of generated inputs should be minimized. For a time budget of five hours, average differences are less pronounced, but the same trends hold.

Table 7. Statistical Significance and Effect Size for Figure 9 and 10

	1 hour budget						5 hours budget					
	p-value			A_{12}			p-value			A_{12}		
S	D	M	A	D	M	A	D	M	A	D	M	A
1	<0.05	<0.05	<0.05	.018	.000	.000	<0.05	<0.05	<0.05	.630	.000	.000
2	<0.05	<0.05	<0.05	.000	.000	.000	<0.05	<0.05	<0.05	.020	.000	.000
3	<0.05	<0.05	<0.05	.039	.000	.000	<0.05	<0.05	<0.05	.093	.000	.000
4	<0.05	<0.05	<0.05	.099	.000	.000	<0.05	<0.05	<0.05	.278	.000	.000
5	<0.05	<0.05	<0.05	.074	.000	.397	<0.05	<0.05	0.81	.090	.000	.490
6	<0.05	<0.05	<0.05	.360	.000	.000	0.15	<0.05	<0.05	.425	.000	.000
7	<0.05	<0.05	<0.05	.117	.000	.000	<0.05	<0.05	<0.05	.123	.000	.000
8	<0.05	<0.05	<0.05	.405	.000	.000	<0.05	<0.05	<0.05	.075	.000	.000
9	<0.05	<0.05	<0.05	.048	.000	.000	<0.05	<0.05	<0.05	.001	.000	.000

Legend: *S*, subject. *D*, comparison with DM2, *M*, Monkey, *A*, APE. We underline the few cases in which statistics indicate that ATUA shows no significant difference (i.e., *p-value* ≥ 0.05) or no higher chances of generating less instructions ($A_{12} > 0.44$) than state-of-the-art approaches.

Table 8 provides p-values and A_{12} statistics. Since we aim to determine if ATUA is likely to generate a higher target instructions/inputs ratio, we look for A_{12} values above 0.50. For a one-hour budget, effect size is always in favor of ATUA (i.e., is more likely to generate a higher instructions/inputs ratio); effect size is always large with respect to Monkey and APE. Even if in a few cases differences are not statistically significant (i.e., we cannot reject the null hypothesis that *there is no difference between ATUA and the state-of-the-art approach concerning the ratio between instructions covered and inputs being triggered*), effect size trends provides a clear picture of the benefits: **ATUA is likely to yield a higher instructions/inputs ratio**. The same conclusions can be drawn for a five-hour budget, though for two subjects (Wikipedia and VLC) ATUA performs similarly to DM2.

To summarize, regarding the human effort required for practical execution time budgets, ATUA performs better than the other approaches, since it saves around 33.8% (1-hour budget) and 32.8% (5-hour budget) of the effort compared with DM2, while it shows huge savings compared to the other two approaches. As for the effectiveness per unit of effort, ATUA provides tangible gains of 68.7% (1 h) and 63.3% (5 h) compared with DM2, and huge differences with the others. **ATUA therefore significantly decreases the human effort required for repairing inputs and defining oracles when compared to state-of-the-art approaches.**

4.4 RQ2: Effectiveness within Time Budget

4.4.1 Experiment Design. To address RQ2, we focus on code coverage results obtained with the updated versions of our subject Apps, i.e., versions V1 to V9. More precisely, we keep trace of the updated methods (hereafter, *target methods*) and instructions belonging to updated methods (hereafter, *target instructions*) that are exercised by the test automation tools considered in our study. To collect data for ATUA and DM2, we rely on the Soot-based code coverage extension integrated into DM2; for Monkey and APE, we rely on MiniTracing, a toolset developed to measure code coverage with APE [28]. Since all these code coverage tools measure the coverage of the whole App under test, to determine the coverage of target methods and instructions, we filter results based on the list of updated methods generated by AppDiff.

Since ATUA and state-of-the-art approaches require a different degree of human effort (see RQ1) and human effort is measured in terms of inputs generated, we shall set an identical limit to the number of inputs that might be generated by the test generation techniques. The rationale is that we try to emulate, in our experiments, realistic conditions where testers are limited by both execution time and human resources. This is thus expected to yield unbiased comparisons of

Table 8. Statistical Significance and Effect Size for Target Instructions/inputs Ratios

	1 hour budget						**5 hours budget**					
	p-value			A_{12}			**p-value**			A_{12}		
S	D	M	A	D	M	A	D	M	A	D	M	A
1	0.10	<0.05	<0.05	.728	1.00	.975	0.69	<0.05	<0.05	.555	.963	.901
2	0.17	<0.05	<0.05	.703	.906	.875	0.14	<0.05	<0.05	.719	.938	.875
3	0.29	0.09	0.09	.700	.820	.820	0.25	0.08	0.17	.720	.840	.760
4	0.33	<0.05	0.05	.636	.895	.772	0.38	<0.05	<0.05	.623	.895	.747
5	0.17	<0.05	<0.05	.691	1.00	.901	0.17	<0.05	<0.05	.691	.950	.827
6	0.63	<0.05	<0.05	.583	1.00	1.00	0.52	<0.05	<0.05	.611	1.00	1.00
7	0.27	<0.05	<0.05	.654	1.00	1.00	0.27	<0.05	<0.05	.654	1.00	1.00
8	0.60	<0.05	<0.05	.578	1.00	.953	0.92	<0.05	<0.05	.516	.968	.906
9	0.39	<0.05	<0.05	.642	1.00	1.00	<0.05	<0.05	<0.05	.914	1.00	1.00

Legend: *S*, subject. *D*, comparison with DM2, *M*, Monkey, *A*, APE. We underline the few cases in which statistics indicate that ATUA shows no significant difference (i.e., *p-value* ≥ 0.05) or no higher likelihood of achieving a higher instructions/inputs ratio (i.e., $A_{12} < 0.56$) than state-of-the-art approaches.

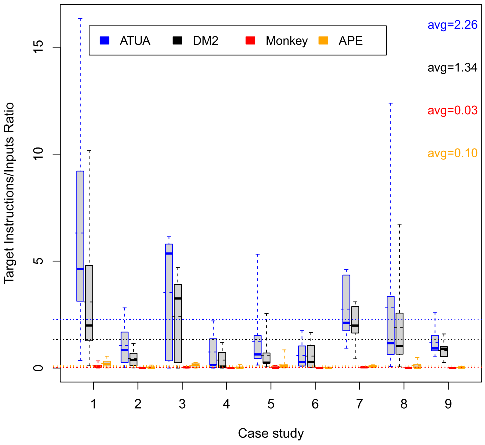


Fig. 13. Target instructions/inputs ratio (budget = 1 hour).

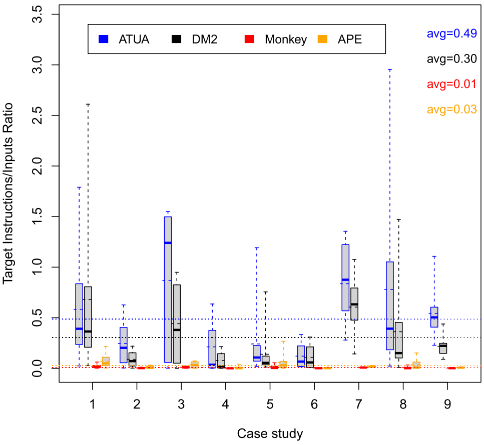


Fig. 14. Target instructions/inputs ratio (budget = 5 hours).

practical value. It is also consistent with our objective, stated earlier, of minimizing human effort while keeping execution time within acceptable bounds. More precisely, for each software version v, we define an inputs budget equal to the maximum number of inputs generated, over 10 runs, by ATUA, which is the approach generating the fewest test inputs for a given time budget, based on RQ1 results.

Though the fault detection rate (i.e., the proportion of faults being detected by a test automation technique) would be a useful, complementary metric to evaluate test automation effectiveness, it is inapplicable in our context, since an important subset of our subject Apps (BBC, YahooWeather, Wikihow, Nuzzel) are not open source, a choice made to include representative Apps. Indeed, the unavailability of source code and bug repositories prevented us from determining if a failure was due to a fault introduced by an App upgrade. Further, supporting the use of coverage for our experiments, it has been recently shown that there is moderate to high correlation between code coverage and the detection of real faults [36]. Finally, without automated functional oracles, in existing studies, effectiveness is typically measured by looking at runtime failures (i.e., due to

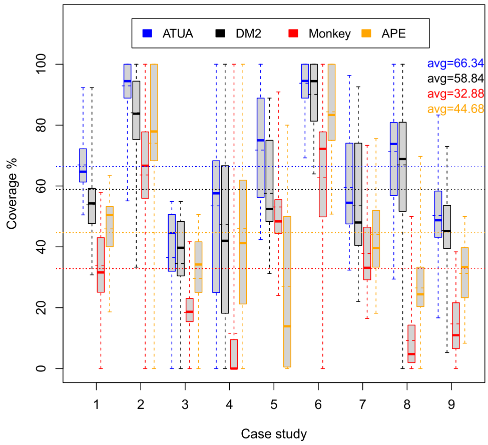


Fig. 15. Percentage of updated methods covered for each version of the case studies (budget = 1 hour).

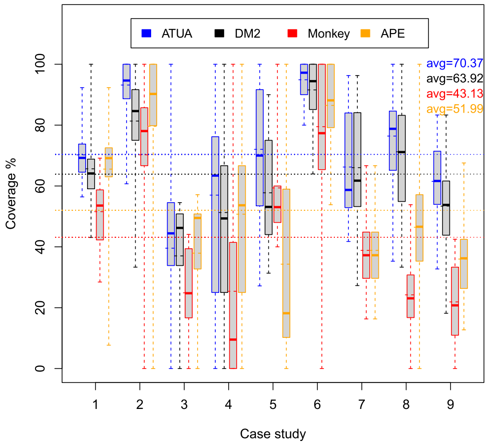


Fig. 16. Percentage of updated methods covered for each version of the case studies (budget = 5 hours).

uncaught exceptions or crashes), which represent only a small proportion of the failures that are typically observed in the field [27].

Metrics. Since the number of target methods and instructions varies across App versions, ATUA and state-of-the-art approaches shall be compared in terms of percentage of target methods and percentage of target instructions covered. In addition, such coverage metrics shall be obtained when testing a subject App for a maximum and practical execution time budget (i.e., one hour and five hours, as discussed in Section 4.2), while not exceeding a maximum input budget determining human effort.

To positively answer this research question, ATUA should, in statistical terms, exercise a larger percentage of target methods and instructions than the other approaches.

4.4.2 Results. Figures 15 and 17 show the distribution of the percentage of target methods and instructions that have been covered by the selected testing tools for the subject Apps, with a test budget of one hour. Figures 16 and 18 report the same measurements for a budget of five hours.

With a test execution budget of one hour, ATUA is the approach with the highest percentage of target methods and instructions being exercised on average, with 66.34% and 56.14%, respectively. The largest differences are observed when ATUA is compared to Monkey; indeed, on average, ATUA exercises 33.46% and 27.42% more methods and instructions than Monkey, respectively. Since Monkey implements a pure random exploration strategy, our results show that a limit on the number of inputs generated by Monkey highly affects its performance. In contrast, the APE state abstraction function enables a more effective generation of test inputs, thus leading to, on average, higher coverage than Monkey. However, ATUA outperforms APE; indeed, on average, ATUA exercises 21.66% and 18.41% more target methods and instructions than APE, respectively. Though DM2 fares better than Monkey and APE, as it relies on a model-based approach leveraging dynamic analysis, ATUA exercises 7.50% and 6.37% more target methods and instructions. Note that the increase achieved by ATUA is particularly significant, +12.74% (i.e., +7.50%/58.84%) and +12.79% (i.e., +6.37%/49.77%) for methods and instructions coverage, respectively. This is explained by the transition-driven exploration based on static analysis (ATUA Phase 1 and 2) and information retrieval (Phase 3), which are not part of DM2.

When executed with a test budget of one hour, for all the subject Apps, both the median and the average obtained with ATUA are higher than those obtained with other approaches.

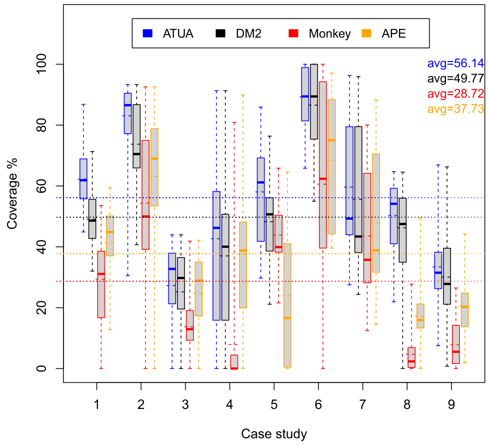


Fig. 17. Percentage of instructions belonging to updated methods that are covered for each version of the case studies (budget = 1 hour).

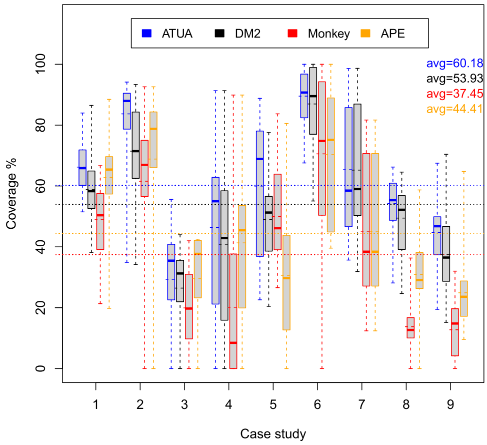


Fig. 18. Percentage of instructions belonging to updated methods that are covered for each version of the case studies (budget = 5 hours).

To discuss differences across subjects, we report in Table 9 the p-value and A_{12} statistics obtained with the Mann Whitney U-test and Vargha and Delaney's method, respectively. Overall, differences are statistically significant[8] but there are exceptions: when ATUA is compared to APE for Nuzzel (subject 4), and when ATUA is compared to DM2 for Nuzzel, File Manager (subject 3), Wikihow (subject 6), and VLC (subject 8). However, for most of the subjects, ATUA is likely to exercise more target methods and instructions than other approaches; this is shown by the A_{12} statistics being always above 0.56, except for File Manager, Wikihow, and VLC in the case of DM2.[9] Regarding VLC, the effectiveness of ATUA is limited by the need for setup operations that require some human effort. Indeed, since certain features can be tested only on specific devices (e.g., an Android TV), identifying target methods through static analysis is of limited usefulness and ATUA performs similarly to DM2. However, such limitations could be surmounted after investing some effort to carefully set up ATUA. For example, by configuring ATUA to be executed on an Android TV in addition to a mobile emulator (i.e., what we used in our experiments). Concerning File Manager and Wikihow, ATUA is affected by some limitations of static analysis, which cannot determine that certain WindowTransitions are associated to specific data types provided as input. More precisely, in the case of File Manager, a number of updated features can be exercised only through specific files (e.g., the decompress operation can be executed only with files having ZIP or RAR filename extension). The static analysis currently implemented in ATUA cannot determine that certain features are enabled only in the presence of specific runtime data (e.g., file names) and thus ATUA, similar to DM2, exercises such features only if it accidentally triggers them, thanks to random exploration. A similar but more evident problem occurs also in the case of Wikihow, where static analysis does not identify the WindowTransitions triggered by the inputs sent to WebViews. Indeed, the input handlers executed after sending an input to a WebView (e.g., a click on an anchor) depend on the content of the page (e.g., the file type appearing in the URL of the anchor) and thus cannot be identified by static analysis, which does not process the content of the HTML pages

[8] We can reject the null hypothesis that *there is no difference in the number of target methods and instructions exercised by ATUA and the* ith *state-of-the-art approach.*

[9] Average A_{12} is 0.77 for target methods and 0.76 for target instructions coverage; median is 0.80 for target methods and 0.77 for target instructions coverage.

Table 9. Statistical Significance and Effect Size for RQ2

	One-hour budget						**Five-hour budget**					
	p-value			A_{12}			**p-value**			A_{12}		
C	D	M	A	D	M	A	D	M	A	D	M	A
	Coverage of target methods											
1	<0.05	<0.05	<0.05	.857	.994	.956	<0.05	<0.05	0.14	.703	.935	.564
2	<0.05	<0.05	<0.05	.733	.868	.703	<0.05	<0.05	<0.05	.774	.832	.623
3	0.41	<0.05	<0.05	.548	.795	.684	0.61	<0.05	0.99	.529	.767	.501
4	0.16	<0.05	0.07	.560	.852	.577	0.16	<0.05	0.08	.560	.760	.575
5	<0.05	<0.05	<0.05	.735	.814	.920	<0.05	<0.05	<0.05	.713	.702	.841
6	0.08	<0.05	<0.05	.587	.838	.689	0.2	<0.05	<0.05	.564	.719	.681
7	<0.05	<0.05	<0.05	.614	.836	.762	0.78	<0.05	<0.05	.488	.879	.879
8	0.23	<0.05	<0.05	.554	.997	.972	0.05	<0.05	<0.05	.588	.995	.892
9	<0.05	<0.05	<0.05	.587	.981	.878	<0.05	<0.05	<0.05	.685	.992	.918
	Coverage of target instructions											
1	<0.05	<0.05	<0.05	.853	.991	.936	<0.05	<0.05	0.15	.735	.884	.562
2	<0.05	<0.05	<0.05	.707	.843	.783	<0.05	<0.05	<0.05	.756	.829	.721
3	0.31	<0.05	0.14	.559	.774	.585	0.18	<0.05	0.70	.577	.705	.478
4	<0.05	<0.05	0.07	.600	.861	.579	0.05	<0.05	<0.05	.584	.764	.594
5	<0.05	<0.05	<0.05	.684	.748	.886	<0.05	<0.05	<0.05	.659	.623	.822
6	0.42	<0.05	<0.05	.542	.788	.827	0.47	<0.05	<0.05	.538	.727	.786
7	<0.05	<0.05	<0.05	.598	.735	.695	0.65	<0.05	<0.05	.480	.780	.780
8	0.06	<0.05	<0.05	.586	.999	.984	0.33	<0.05	<0.05	.633	.999	.936
9	0.05	<0.05	<0.05	.584	.980	.822	<0.05	<0.05	<0.05	.671	.988	.896

Legend: *C*, case study; *D*, comparison with DM2; *M*, Monkey; *A*, APE. We underline the few cases in which statistics indicate that ATUA shows no significant difference (i.e., *p-value* ≥ 0.05) or no higher likelihood (i.e., $A_{12} < 0.56$) of covering more targets than state-of-the-art approaches.

displayed at runtime. For this reason, ATUA cannot fully take advantage of static analysis results in the presence of WebViews. In such cases, similar to DM2, ATUA exercises App features, thanks to random exploration. However, current ATUA results with Apps using WebViews largely depend on the proportion of features implemented through WebViews. For example, in the case of WikiHow, which mainly relies on WebViews (five out of eight content types are displayed through a WebView), ATUA performs similarly to DM2; instead, in the case of Wikipedia, which implements only one out of 35 Windows using a WebView,[10] ATUA outperforms all the other approaches ($A_{12} \geq 0.56$). To overcome the limitations of static analysis and thus improve ATUA results, it might be necessary to develop dedicated strategies relying on dynamic analysis; for example, by extending the state abstraction function of ATUA to use reducers dedicated to HTML anchors or file objects.

With a test budget of five hours, all the approaches achieve better coverage results; however, the ranking observed for a one-hour budget remains unchanged. ATUA is the approach with the highest percentage of exercised target methods and instructions, on average, with 70.37% and 60.18%, respectively. The largest differences are still observed when ATUA is compared to Monkey; indeed, on average, ATUA exercises 27.24% and 22.73% more target methods and instructions than Monkey, respectively. ATUA exercises 18.38% and 15.77% more target methods and instructions than APE. The second-best approach remains DM2, as ATUA exercises 6.45% and 6.25% more target methods and instructions than DM2, with a gain of +10.09% (+6.45%/63.92) and +11.59%

[10] In Wikipedia, WebViews are used to display Wikipedia pages, while other Views are used for other features such as displaying news, image galleries, or editing the content of a page.

(+6.25%/53.93%) in the number of target methods and instructions covered with respect to DM2, respectively.

A larger time budget enables ATUA to achieve higher coverage. This is set with the *scaleFactor* configuration parameter (see Section 3.5.3), which we increase for a five-hour budget, thus augmenting the time spent to perform random exploration, reach the test target, and exercise targets. We leave to future work the study of the effect of different configuration values for the *scaleFactor* parameter of ATUA.

With a test budget of five hours, the difference between ATUA and other approaches decreases, though. Unsurprisingly, with a larger test budget, random-based approaches can more easily reach updated features than with a one-hour budget, for which leveraging static analysis is more important. For example, in subject App 7 (BBC Mobile), in five hours, DM2 achieves the same average coverage as ATUA.

To discuss differences across subjects, we refer to the p-value and A_{12} statistics reported in the rightmost columns of Table 9. Because of the larger test budget benefiting random exploration, differences between ATUA and other approaches are not significant in 7 out of 27 cases (i.e., 3 × 9, which is the number of pairwise comparisons between ATUA and the other approaches), two cases more than with a one-hour budget. However, ATUA is still likely to exercise more target methods and instructions than other approaches. Indeed, for both method and instruction coverage, the A_{12} statistics are above 0.56 for 24 out of 27 cases. In general, effect size is slightly lower than for a one-hour budget, with an average A_{12} of 0.73 and 0.72 for the coverage of target methods and instructions. In particular, we observe that the larger time budget enables random-driven approaches to achieve the same effectiveness as ATUA when ATUA is negatively affected by static analysis limitations. This happens for File Manager (subject 3), where APE performs similarly to ATUA, Wikihow (subject 6), where DM2 performs similarly to ATUA, and BBC mobile (subject 7), where the additional time budget enables DM2 to exercise the few updated features depending on WebViews (in BBC Mobile, WebViews are used to display BBC Web pages).

To summarize, ATUA is the approach that, on average, most effectively test updated Apps within practical time budgets and human effort. It tends to cover more target methods and instructions than other approaches. The second-best approach is DM2. For a one-hour budget, on average, ATUA automatically exercises 7.50% and 6.37% more target methods and instructions than DM2, with a gain of +12.74% and +12.79%, respectively. With a five-hour budget, on average, ATUA automatically exercises 6.45% and 6.25% more target methods and instructions than DM2, with a gain of +10.09% and +11.59%, respectively. For seven out of nine subjects, for both time budgets, ATUA tends to exercise more target methods and instructions than DM2. For the remaining two subjects, DM2 and ATUA are comparable, due mostly to the current limitations of static analysis.

4.5 RQ3 - Complementarity of Testing Approaches

4.5.1 Experiment Design. A software testing approach is complementary to other approaches if it exercises a set of functionalities not exercised by the others. Since we measure effectiveness based on method coverage, to determine complementarity, we look for methods that are univocally covered by each testing approach considered in our experiments. A method is *univocally covered* by approach A for version V of a subject App S if it is exercised by A in at least one of the 10 test execution runs on version V and is not exercised by any other approach in any test execution run of that same version. We cannot compare testing approaches based on instruction coverage, because some of our subjects are commercial Apps released without source code (e.g., to understand the semantics of the covered instructions). Since the number of target methods varies for each App version, we compare the percentage of target methods that are univocally covered by each

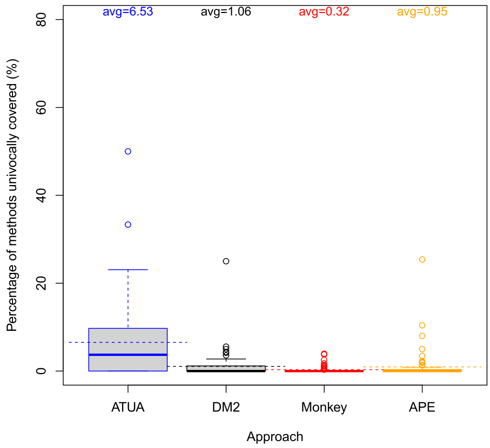


Fig. 19. Proportion of methods that are univocally covered by one testing approach, distribution across all the tested subject versions (budget = 1 hour).

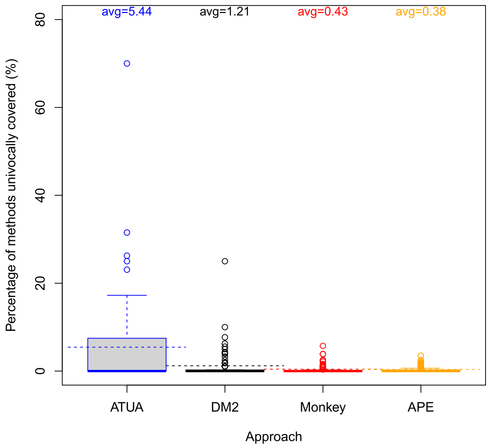


Fig. 20. Proportion of methods that are univocally covered by one testing approach, distribution across all the tested subject versions (budget = 5 hours).

approach. Finally, it shall be possible to identify common characteristics in the inputs triggering the univocally exercised methods.

Metrics. To compare testing approaches, we thus report (1) the *overall number of univocally covered methods across all the subject App versions* and (2) *the distribution of the percentage of tested methods that are univocally covered by each approach, across all the subject App versions.* Furthermore, we manually inspect the list of univocally covered methods. Based on their signatures[11] and, for ATUA, the data collected in the GSTG, we (3) *determine the characteristics of the inputs and upgraded functionalities that are better targeted by each of the testing approaches.* Since the test budget affects the performance of testing approaches, we discuss the results achieved for one-hour and five-hour budgets, separately.

For each testing approach, we analyze if (1) it covers a large number of methods not covered by other approaches across all Apps, (2) its distribution, across Apps, of the percentage of tested methods not covered by other approaches has a significantly larger average than that of other approaches, and (3) we can characterize the situations in which the approach univocally exercises some updated App methods.

4.5.2 Results. For the one-hour budget, a total of 6,982 methods belonging to the different App versions have been exercised in our experiments. Overall, 784 out of 6,982 methods (11%) are exercised only by one testing approach, 6,198 methods (89%) are covered by at least two approaches, while 3,648 methods (52%) are covered by all the approaches. ATUA exercises the largest number of methods not exercised by other approaches, 518 (66%); it is followed by APE (156, i.e., 20%), DM2 (78, i.e., 10%), and Monkey (32, i.e., 4%).

Figure 19 shows the distribution of the percentage of methods exercised in our experiments that are univocally covered by one testing approach, across versions, for a one-hour budget. On average, 7% of the methods exercised in our experiments are covered only by ATUA, while the other approaches univocally cover only 1% or less. Differences are statistically significant. Also, we report that for 52% of the App versions, ATUA exercises more univocally covered methods than other

[11] Since most of our subject Apps are commercial Apps released without source code, the functionality implemented by a method is inferred from its signature.

approaches (92% if including versions with the same number of univocally covered methods). These results show that, across a majority of individual versions, ATUA provides coverage capabilities that cannot be obtained with other approaches.

The effectiveness of ATUA is primarily due to its capability of reaching target Windows and target Widgets that are difficult to reach by solely relying on random exploration. We identify three distinct cases. *First*, ATUA can trigger complex sequences of inputs that enable the visualization of target Widgets. This is the case of SettingsActivity for Nuzzel, which requires opening a drawer, swipe up, and then click on the settings button. Similarly, in Yahoo Weather, it is necessary to swipe up the weather information fragment and click on the map to trigger methods on the WeatherMapView. Other similar cases concern the renaming of files and the opening of the preferences activity in File Manager. Such complex sequences of events are unlikely to be triggered by approaches relying on random exploration; instead, they are selected by ATUA, thanks to the use of the App model to identify both the sequence of events that reaches a target Window and the target events that exercise the updated methods. The *second* case concerns ATUA being able to bring an App into a specific state required for testing, which is enabled by the fact that, in Phase 2, ATUA exercises the inputs that trigger updated methods multiple times, when the App has likely reached different App states. For example, ATUA is the only approach exercising the method *undeleteActivity* of class *ActivityHelper* in Activity diary, which requires to first create an activity, then delete it, and finally undelete it. The *third* case concerns ATUA's capacity to select the Apps' options required to exercise certain Apps' features, which is the objective of Phase 3. For example, in BBC mobile, to exercise the methods in class *MyNewsByTimeFragment*, it is necessary to reach the settings window, enable the option *My News By Topic*, and then open the tab *My News*.

The methods not covered by ATUA but covered by other approaches, instead, are generally the ones whose triggering Actions cannot be identified by ATUA because of the limitations of static analysis. We have identified three different scenarios in which ATUA is less effective than state-of-the-art approaches. First, though ATUA can trigger complex input sequences, it cannot, in certain cases (e.g., in classes extending SettingsActivity), identify the events that trigger specific WindowTransitions, which APE can instead trigger. By relying on an adaptable state abstraction function, APE can direct random exploration towards App states that contain the widgets required to test the updated methods, which Monkey and DM2 do not achieve. For example, this happens when testing the updated methods of *AboutActivity* in File Manager. Second, in the case of WebViews, instead, DM2, by investing more budget on random exploration, can reach Windows that cannot be reached by ATUA, because static analysis does not identify the required WindowTransitions in the EWTG. Also, based on the observed results, the specific random exploration strategy implemented by DM2 appears to be more effective than the one of APE and Monkey. Finally, Monkey performs better than ATUA and the other approaches when the execution of updated methods depends on specific environmental conditions; for example, the internet connection being disabled, which is the case for testing methods *onGoOffline* and *onPageLoadError* in Wikipedia.

Similar findings can be observed for a test budget of five hours. Overall, a total of 7,326 methods have been exercised in our experiments, which is expectedly higher than for the one-hour budget. Overall, 675 out of 7,326 methods (9%) are exercised only by one testing approach, 6,767 methods (92%) are covered by at least two approaches, while 4,308 methods (58%) are covered by all approaches. Although the larger test budget enables all the approaches to exercise a larger common set of methods, we still observe a high degree of complementarity (i.e., 9% of the covered methods are univocally covered). In particular, ATUA remains the approach that exercises the largest number of methods not exercised by other approaches: 478 (71%); the other approaches, instead, show similar numbers of univocally covered methods: 71 for APE, 64 for DM2, 62 for Monkey.

Figure 20 depicts the distribution of the percentage of covered methods that are univocally covered by one testing approach, across versions, for a budget of five hours. On average, across versions, 5% of the methods exercised in our experiments are covered only by ATUA, while the other approaches univocally cover only 1% or less, thus confirming that, even for a test budget of five hours, ATUA complements all the other approaches. Differences are statistically significant. Such complementarity is also stressed by the fact that for 42% of the App versions, ATUA covers more univocally covered methods than other approaches (81% if including versions with the same number of univocally covered methods).

Also, for a five-hour budget, ATUA confirms its capacity to trigger complex sequences of inputs not generated by other approaches. This is the case for File Manager, where ATUA successfully starts the FTP client by (1) clicking on the "add" button, (2) then clicking on "Cloud connection," (3) then clicking on SCP/SFTP connection, and (4) finally, within the SCP/SFTP connection dialog, filling all the compulsory fields and (5) clicking on the "Create" button. Such a complex sequence of inputs (including filling FTP connection information) is unlikely to be generated by random approaches. ATUA, instead, once it finds the sequence of inputs that reaches the SCP/SFTP connection dialog, can, in Phase 2, trigger multiple sequences of inputs until it (randomly) finds the one that successfully starts the FTP client. For APE, Monkey, and DM2, we can observe the same characteristics observed as for a one-hour test budget.

To summarize, for both one-hour and five-hour budgets, **ATUA is the approach that exercises the largest number of univocally covered methods**. Across versions, the percentage of exercised methods univocally covered by ATUA is significantly larger than that of other approaches. In practice, the results above also suggest that it might be useful to combine approaches, since they may complement each other to cover a larger number of methods. However, ATUA should always be included in the selected combination, since it exercises a larger set of upgraded functionalities that cannot be exercised using other approaches.

4.6 Discussion

Human effort. RQ1 results have shown that ATUA performs better than the other approaches, since it saves around 33.8% (one-hour budget) and 32.8% (five-hour budget) of the effort compared with DM2, the second-best approach. Hereafter, we discuss practical implications concerning testing costs based on related work about the nature of App upgrades [44] and the maintainability of GUI test cases [48].

On average, ATUA generates 450 (one hour) and 2,251 (five hours) fewer inputs than DM2 for each App version across all subject Apps. Since related work [44] has shown that roughly 35% of the updates concern the introduction of new features, under the assumption that inputs are uniformly distributed across updated features, we can estimate that ATUA generates, on average for each App version and across all subjects, 158 (one hour) and 788 (five hours) fewer inputs than DM2 for testing new features. Consequently, ATUA generates 292 (one hour) and 1,463 (five hours) fewer inputs than DM2 for testing bug fixes and improved features (i.e., changes concerning non-functional requirements).

When testing new features, the output generated by each input should be manually verified; for example, by inspecting the screenshots of the GUI trees visualized after triggering an input (they are automatically captured by ATUA) and determining if they match the expected results. Unfortunately, the software engineering literature lacks studies about the cost of manual verification of GUI trees; assuming, for the sake of illustration, that visual inspection of GUI trees takes a few minutes, say, ranging from one minute to five minutes, ATUA may lead to savings within the following intervals of [158–790] and [788–3,940] minutes, respectively, the for one-hour and five-hour test budgets. In the App development context, where Apps are frequently released (e.g., weekly or bi-weekly) and, additionally, test cases might need to be executed every day following continuous integration practices, such effort savings appear to be particularly beneficial, especially considering that testing should be performed by highly trained engineers with a deep understanding of the App's features.

When testing updated features, engineers can re-execute the generated test input sequences on previous App versions to compare results and, ideally, eliminate oracle costs. However, we have to expect that a number of maintenance operations are required to adapt test sequences to a different App version. Pan et al. [48], for example, report that 26.5% of the test inputs need to be repaired. ATUA will thus save engineers from manually repairing 77 and 388 inputs, respectively, for the one-hour and five-hour test budgets. Under time pressure, which is the case when Apps are frequently released, this is a significant advantage.

Effectiveness. ATUA is the approach that, on average, most effectively tests updated Apps within practical time budgets and human effort. For the one-hour budget, better than competing approaches, it exercises more than 60% of target methods and 50% of target instructions. With a five-hour budget, it exercises more than 70% of target methods and 60% of target instructions. Higher percentages can probably be reached with longer execution budgets, which were not possible in our context given the computational costs of our experiments. Based on these results, we can claim that ATUA can contribute to reducing development costs; indeed, engineers would then be able to focus their manual testing effort on a reduced portion of the developed App.

When comparing with other approaches, we observed that for both one-hour and five-hour budgets, on average, ATUA achieves method and instruction coverage results increased by at least 10% with respect to the second-best approach (DM2), a practically significant improvement. The effectiveness of ATUA is comparable to the effectiveness of DM2 and APE only when ATUA cannot fully leverage static analysis to determine the relation between inputs and WindowTransitions, i.e., when Apps integrate input handlers that are selected at runtime based on the nature of input data, which happens, for example, in the presence of WebViews. For the six subjects for which static analysis can effectively be exploited, the percentage of improvement rises above 8%. Among our subject Apps, in the worst case (i.e., five-hour budget), ATUA is comparable to other approaches for one-third of the subjects and otherwise fares better; considering that (1) no single competing approach achieves similar coverage as ATUA for these three subject Apps (e.g., DM2 achieves the same results as ATUA for at most two), (2) competing approaches never outperform ATUA but at best reach the same effectiveness, ATUA remains the best choice. To further improve ATUA effectiveness, part of our future work concerns the development of an additional set of reducers that will enable the ATUA state abstraction function to distinguish between widgets containing different types of data.

Finally, ATUA has shown to be complementary to other approaches. Indeed, for both one-hour and five-hour budgets, it can exercise 518 and 478 target methods not covered by other approaches, three and six times the number of the second-best approach. Thus, when combining testing approaches to cover higher target method coverage, ATUA should be included. Finally, as an explanation of the above results, we have observed that the three testing phases integrated in ATUA enable the generation of complex input sequences, specific App states, or diverse App settings that are required to test updated features.

4.7 Threats to Validity

We discuss internal, conclusion, construct, and external validity according to standard practice [24, 53, 74].

4.7.1 Internal Validity. To address threats to *internal validity*, we should ensure that the observed outcome (inputs and code coverage, in our case) depends on the treatment (i.e., the test automation approaches) and not external factors (e.g., implementation errors and diverse experimental conditions) [24].

To minimize *implementation errors*, we have carefully inspected and tested ATUA before running our experiments. Also, for the state-of-the-art approaches, we relied on the software released by their authors, which had been used in several experiments.

To *ensure the same conditions* for all the experiments, we executed each tool on a clean instance of the same Android emulator with newly installed Apps. However, the same experimental conditions may not be guaranteed for Apps that depend on external data sources (e.g., to visualize news) [19]; indeed, in the presence of external data source, test results may depend on the content being visualized at a specific instant (e.g., the presence of a video in the latest news). Our case study subjects include six Apps loading external data (i.e., Nuzzel, Wikipedia, Yahoo weather, Wikihow, BBC Mobile, and Citymapper), because they are highly popular and representative.

To address this threat, taking advantage of our Grid infrastructure, for each test budget (i.e., one hour and five hours), for each subject App version, we executed all the testing tools in parallel in five batches with two sequential executions each. In practice, for each subject App's version, for each tool, we ran 10 executions distributed over a time frame of two hours (for a one-hour budget) and 10 hours (for a five-hour budget). Our experimental configuration should minimize the threat for Apps (i.e., Wikipedia, Wikihow, and Citymapper) loading remote content that unlikely changes in the time frame of our executions (i.e., 10 hours max). In addition, we believe that our configuration also addresses the threat for the remaining three Apps (i.e., Nuzzel, Yahoo weather, BBC Mobile) because, by running all the different testing tools in parallel, we maximize the likelihood of processing the same remote content (i.e., news or weather forecasts) when triggering the same Actions.

4.7.2 Conclusion Validity. Threats to *conclusion validity* concern the statistical power of our results, invalid statistical test assumptions, reliability of measurements, and random irrelevancies [74].

Since the underlying distribution of the data (i.e., code coverage achieved with test automation approaches) is not known in our context, for statistical significance, we rely on the Mann Whitney U-test, which has high *statistical power* for different underlying distributions, even for a small number of samples [62]. Also, to let the readers draw conclusions in context about the proposed approach, we report both p-values and effect sizes.

To avoid violating the *assumptions of parametrical statistical tests*, we rely on a non-parametric test and effect size measure (i.e., Mann Whitney U-test and the Vargha and Delaney's A_{12} statistics, respectively).

To ensure *reliability*, our measurements (i.e., code coverage) have been collected through widely used, open-source tools.

In our context, the only source of *random irrelevancies* might be the workload of the machines used to run the experiments, which may slow down the performance of some of the tools. To mitigate this threat, in addition to relying on a Grid infrastructure with guarantees for the provided service level, we manually inspected execution logs to exclude the presence of anomalies biasing results (e.g., exceptions due to the host environment).

4.7.3 Construct Validity. According to standard practice, we discuss *construct validity* in terms of face, content, convergent, and predictive validity [53]. The constructs considered in our work are effectiveness and cost. Effectiveness is measured through two reflective indicators, which are target method coverage and target instruction coverage. Cost is measured in terms of the number of inputs being generated, for reasons that were carefully discussed.

Face validity concerns the selection of appropriate reflective indicators. For effectiveness, we rely on method and instruction coverage, which is common practice [16, 19]. For cost, we measure the number of inputs being generated. At the beginning of Section 4, we have discussed that, in our context, the number of inputs is a good surrogate to enable the comparison of testing cost.

Content validity concerns the adequacy of reflective indicators to cover the breadth of the construct. We rely on code coverage, since it has been recently shown that there is moderate to high correlation between code coverage and detection of real faults [36]. Also, code coverage is a necessary condition to uncover faults and, therefore, it remains a priority for test engineers. Concerning the breadth of the cost construct, in the introduction to this section, we have discussed that a direct and precise cost estimate can only be obtained with experiments involving engineers using the selected testing techniques in the field under controlled conditions, which we leave to future work.

Concerning *convergence*, we have computed the non-parametric Kendall's correlation coefficient, for all the pairs of reflective indicators, for each subject. Unsurprisingly, target method and target instruction coverage are highly correlated (i.e., $\tau \geq 0.7$, for all the subjects), which is expected for reflective indicators used to infer the same construct. Instead, a low correlation (i.e., $\tau < 0.35$, for all the subjects) is observed between inputs being triggered and target coverage, which is expected, since these two reflective indicators are used for distinct constructs.

To address *predictive validity*, we reported statistics for all our research questions.

4.7.4 External Validity. To address threats to *external validity*, we have considered nine popular Apps, downloaded thousands of times worldwide, that have been considered in the empirical evaluation of related work. Also, for each App, we considered up to 10 App versions, based on their availability, for a total of 72 App versions tested. The considered Apps greatly vary regarding the overall number of lines of code and updated lines between versions. Because of their diversity, we believe our subjects to be representative of the Apps landscape.

To account for randomness, we tested each App version 10 times with every testing tool considered; more than the usual practice of three to five repetitions [29]. Despite the high computational cost (17,280 test execution hours, in total), this enabled us to derive solid statistical results for the comparison of different tools.

In our experiments, we considered only Android Apps, which is standard practice in most App testing research papers. The prevalence of Android in research papers is mostly due to its worldwide dissemination and the availability of a larger set of tools to test and analyze Android executable bytecode [22]. In our work, the choice of relying on Android Apps enabled the comparison of ATUA with tools working for Android Apps (i.e., Monkey, APE, and DM2). However, since we do not exercise OS-specific features, results should generalize also to Apps running in different execution environments (e.g., HarmonyOS and IOS).

5 RELATED WORK

In this section, we discuss related work, which covers automated App testing tools, App regression testing techniques, incremental testing approaches, testing based on information retrieval, and test oracle automation.

5.1 App Testing Tools

Automated App testing tools can be grouped according to the strategy adopted to generate test inputs [41, 67]. The most common ones are random, model-based, and evolutionary [67].

Representative approaches of these three categories used in empirical evaluations are Monkey, Stoat [65], and Sapienz [43], respectively. Monkey has been introduced in Section 4. Other recent and effective approaches either rely on Q-Learning [38, 46], execution traces [69], or state cloning [23].

Stoat performs stochastic model-based testing. It relies on dynamic analysis based on a weighted UI exploration strategy to derive a stochastic **finite state machine (FSM)** of the App's GUI interactions. Stoat relies on the FSM to generate test suites using an objective function that aims to maximize code coverage, model coverage, and test diversity. The test generation process relies on Gibbs sampling to iteratively mutate and refine the FSM, based on the fitness of the generated test suite.

Sapienz is an evolutionary approach that uses Pareto multi-objective search to automatically explore and optimise test sequences, minimizing length, while simultaneously maximizing coverage and fault detection. Sapienz combines random fuzzing, systematic exploration, and search-based exploration.

Independent empirical evaluations performed by Choudhary et al. [16] and Wang et al. [70] have reported that the three aforementioned testing strategies are complementary. Further, both studies show that the method and instruction coverage achieved by all test automation approaches are relatively low, that is, below 50%. Choudhary et al. [16] note that model-based approaches complement random approaches regarding fault detection, while for code coverage, random approaches fare better. Wang et al. [70] confirm these results. They report that random and evolutionary approaches are complementary regarding method coverage, while both evolutionary and model-based approaches complement random approaches in terms of fault detection. However, the validity of these findings has been weakened by recent advances in model-based approaches. Indeed, more recent results show that model-based approaches that either integrate advanced exploration strategies (i.e., biased random in DM2) or adaptable state abstraction functions (i.e., APE [29]) fare better than random approaches or state-of-the-art model-based approaches. APE, for example, is the most recent technique and has been reported to perform better than Monkey, Sapienz, and Stoat.

In *APE*, each window is modeled with sets of *attribute paths* that univocally identify the widgets of the window. Attribute paths resemble the AttributeValuationMaps of ATUA with the difference that ATUA considers a larger set of attributes than APE, which only accounts for type, position with respect to siblings, and appearance (e.g., text). APE and ATUA differ for the strategy used to generate inputs (i.e., ATUA relies on the combination of static and dynamic analysis). They both rely on an adaptable state abstraction function $\mathcal{L}$. However, the state abstraction functions integrated in the two approaches present key differences. In APE, a single $\mathcal{L}$ is defined for the whole app, while ATUA specifies one $\mathcal{L}$ for each window of the App. Also, APE's $\mathcal{L}$ is implemented by means of a **decision tree (DT)**, which enables APE to not rely on a predefined set of reducer functions. The main limitation of APE is that it relies on a global abstraction function for the whole App and thus uses part of the test budget to perform $\mathcal{L}$ refinements that may be avoided by relying on static analysis (i.e., GUITrees belonging to different windows should be characterized by different abstract states). By relying on a different $\mathcal{L}$ for every window, ATUA overcomes such limitation. In addition, our empirical evaluation has shown that the combination of static and dynamic program analysis enables ATUA to outperform APE in terms of coverage of updated methods, while minimizing the human effort required by testing.

Approaches not relying on random, model-based, or evolutionary solutions make use of either Q-Learning [38, 46], execution traces [69], or state cloning [23]. *Q-testing* [46] relies on Q-learning (a model-free reinforcement learning algorithm [72]) to trigger events that enable the exploration of features that are not yet tested. In Q-testing, an event is rewarded if it leads to an App state not visited before. A siamese neural network is used to compare states. *QBE* is a less recent approach based on Q-learning, which relies on automated App exploratory testing to identify App states [38]. We did not select Q-testing [46] for our empirical evaluation, because there is no empirical evidence demonstrating it outperforms APE. For QBE, however, existing evidence shows it does not outperform state-of-the-art approaches. *Combodroid* works by combining test input sequences derived from execution traces [69]. It can work with either traces collected by automated testing tools (e.g., APE) or traces collected when end-users exercise the App under test. In the first case, results show that Combodroid achieves higher code coverage than APE when using more than six hours of test budget, which is not feasible in continuous integration contexts where test cases are always executed after code commits. The need for execution traces collected with humans in the loop makes the second usage scenario of Combodroid inapplicable in our context. For these reasons, we did not compare ATUA with Combodroid. *TimeMachine* implements a metaheuristic approach that relies on a pool of App states [23]. New App states are reached by triggering random events. An App state is added to the pool only if it is reached after exercising code not covered yet; App states are captured by cloning the state of the virtual machine running the App under test. When lack of progress is detected, TimeMachine resets the execution from the state with the highest fitness, which is computed by balancing the number of times the state has been visited and the number of interesting states generated from it. Similar to Combodroid, TimeMachine overcomes Monkeys, Stoat, and Sapienz when executed for more than five hours; however, as discussed above, this setting makes it inapplicable in some continuous integration contexts. For the reasons above, we did not consider ComboDroid, Q-testing, and TimeMachine to be suitable candidates for our empirical evaluation.

Most importantly, none of the existing App testing approaches prioritize the testing of App updates. **ATUA is the first solution addressing App updates** by focusing on updated methods. To efficiently test updated methods, ATUA integrates a model-based approach that combines static and dynamic program analyses. In the App context, static analysis is the most appropriate solution to efficiently identify test inputs, because (1) Apps architecture enables the extraction of models capturing window transitions through static analysis, and (2) call graph analysis enables the identification of the inputs that trigger event handlers reaching modified code. Dynamic analysis, instead, enables overcoming static analysis limitations; for this reason, ATUA associates abstract states derived through model-based exploration (i.e., dynamic analysis) to windows in EWTGs derived with static analysis.

Alternative potential solutions, based on augmenting exiting approaches, can be considered to test updated Apps. For example, APE might be extended to spend more test budget on states triggering updated methods, Q-testing might be extended by increasing the reward for updated methods, ComboDroid by increasing the chance of exercising use cases related to changed classes, while TimeMachine's fitness could be adjusted to prioritize change-related states. However, such solutions will unlikely reach updated code efficiently if they are not combined with the innovative contributions provided by ATUA, that is, procedures to combine information collected with static and dynamic program analysis.

To test updated Apps, APE needs to be extended with the Phase2 and Phase3 strategies implemented by ATUA. More precisely, APE natively works by prioritizing inputs (called model actions in APE) not exercised in an App state, and extending APE to prioritize target inputs based on static program analysis data (e.g., the list of target inputs generated by Extended gator, one of our contributions) may not be sufficient to efficiently test upgrades. Indeed, based on our results, when testing an updated App, it is often necessary to exercise a same Window multiple times with a same target input to reach the desired abstract state (this happens in ATUA's Phase 2, as shown in our running example); also, it may be necessary to exercise related windows to cover a target method (this happens in ATUA's Phase 3, and it makes ATUA more effective than related approaches, as discussed in RQ3). Considering that APE does not even track coverage information (e.g., what inputs increase coverage) this would basically lead to a re-implementation of ATUA with a different state abstraction function (i.e., the one used by APE, which, however, does not support integration with static analysis). Concerning ComboDroid, in addition to static program analysis, it would be necessary to extend it with a dynamically refined state abstraction function like the one implemented by ATUA. Indeed, for both automatically and manually derived use cases, ComboDroid relies on a static state abstraction function to transform a sequence of events (automatically generated or manually triggered by engineers) into an extended labeled transition system that is similar to ATUA's DSTG. However, ComboDroid's state abstraction function implements a transformation that is similar to the one achieved with ATUA's abstraction level L1, that is, it ignores text content and children widgets. As shown in our running example, which is based on our case study subjects, a dynamically refined state abstraction function is necessary to reach the abstract states required to exercise updated features. Moreover, in the App upgrade context, where new features may be introduced into the App under test, the need for traces manually recorded by engineers may largely limit the benefits of test automation; indeed, to record such traces, engineers may need to exercise all the features introduced into the App under test, that is, manually perform system-level testing of the App, which is what we aim to automate with ATUA. TimeMachine, similar to APE, would also need to integrate the ATUA's strategy for the selection of test inputs; an alternative is to extend ATUA to rely on TimeMachine's time travel features to quickly reach abstract states previously visited (e.g., at the beginning of testing); we leave it for future work. Q-testing, instead, is unlikely to achieve the results obtained with ATUA's Phase 3; indeed, without a predefined strategy to select related windows it is unlikely to automatically discover them by means of trial and error (i.e., what Q-learning does in its training phase). Indeed, as shown in our discussion for RQ3, randomly generated inputs do not enable test automation tools (i.e., Monkey, APE, and DM2) to reach some of the updated methods that need to be enabled in Windows different than the one with a target Widget.

To summarize, state-of-the-art techniques are unlikely to achieve ATUA's performance by augmenting them with simple input selection strategies based on static program analysis (e.g., prioritize inputs that may trigger modified methods, based on static program analysis); also, they cannot be efficiently integrated with all the solutions provided by ATUA: (1) APE does not track code coverage, which is required to drive testing, nor has a state abstraction function been integrated with static program analysis. (2) In our context, the manual recording of traces required by ComboDroid requires manual testing of the updated App features (i.e., what ATUA automates). (3) Q-testing is based on a paradigm (i.e., reinforcement learning) that cannot make use of a static analysis model and cannot be integrated with the three ATUA's phases. (4) The time travel feature proposed by TimeMachine might be integrated into ATUA but, since it has shown to be effective, such integration may be the focus of future work.

5.2 App Regression Testing Techniques

Regression testing techniques for Apps concern the selection of events that may trigger modified code [61], the selection of regression test cases [15], and the repair of existing test suites [63]. QADroid is a static analysis toolset that identifies the events (i.e., the Inputs of ATUA EWTG) that may trigger the execution of modified methods. It implements the features of Steps 1 and 2 of ATUA, except that QADroid does not generate EWTGs. QADroid differs from ATUA regarding the underlying static analysis framework used to identify Inputs. It relies on FlowDroid [8], while ATUA relies on Gator because of its capability to generate WTGs. Also, empirical results show that Gator performs better than FlowDroid in identifying valid sequences of callbacks [71]. Different from ATUA, to select the Inputs that may trigger modified methods, QADroid performs a forward traversal of the App control flow graph obtained with Soot (i.e., it starts the traversal from event handler methods) and selects all the Inputs that reach modified methods. Consequently, QADroid cannot determine the presence of *HiddenHandlers* identified by ATUA. In addition, QADroid requires the source code of the App, while ATUA works with the bytecode. QADroid has never been applied to automated testing and cannot identify the concrete input values to be used with certain Inputs (e.g., the data to write into a TextArea).

Redroid selects a regression test suite for an updated version of an App [20, 21] from an existing test suite. It relies on state-of-the-art static analysis procedures [55] to perform change impact analysis and uses code coverage information to select any test case that cover modified blocks of code. Redroid does not support the generation of a minimal test suite. *DetReduce* [15], instead, creates a small regression test suite for an App from a test suite generated by a model-based test automation tool. It identifies and removes redundant method call traces and subtraces within the test suite and redundant loops within a test case. Redundancy is identified based on a state abstraction function that considers actionable widgets and their visible attribute values. Widgets are identified by their path in the GUITree. Since executable test suites are often unavailable and state-of-the-art App testing tools can cover only a narrow set of modified methods, the applicability of Redroid and DetReduce remains limited. However, *DetReduce* might be applied to ATUA App models to further reduce the generated test cases; we leave to future work the integration of ATUA with DetReduce and its evaluation. Another solution for the generation of reduced test suites is the one proposed by Jabbarvand et al. [33] in the context of energy-aware test-suite minimization. In their work, they stop and restart the execution of Monkey every 500 events; a sequence of 500 events is a test case. When the test budget is exhausted, a greedy algorithm identifies the minimal set of test cases that maximize the coverage of energy-greedy parts of the App. Such solution might be adapted to work with ATUA and the state-of-the-art approaches considered in our experiments; for example, by restarting test generation after a predefined number of events and then by relying on the greedy algorithm to select the minimal set of test cases (e.g., up to 2,000 inputs) that maximizes the coverage of updated methods. However, though such an approach might reduce the size of the generated test suites, it might not be feasible to identify a configuration that maximizes the benefits for a range of Apps. Indeed, for some Apps, 500 events might not be sufficient to reach an App state that enables exercising updated methods, especially for random approaches (e.g., if updated methods require long, specific sequences of events to be exercised), while a much larger number of events would greatly reduce the benefits of test suite reduction (e.g., for one-hour budget, the test cases generated by ATUA consist of less than 2,000 events; see Figure 11). We therefore leave to future work the integration of such test suite minimization solution into ATUA.

Automated repair techniques for the GUI test scripts of mobile Apps are still preliminary [47, 48, 63]; to repair test scripts they include strategies ranging from static program analysis [63], to model-based [40] and computer vision techniques [47, 48]. Existing approaches either leave between 5% [48] and 8% [63] of the test scripts to be manually repaired or do not preserve all the test actions (i.e., the test semantic [40]). Though these results show that automated GUI script repair techniques might be adopted to support the oracle automation approach we suggested for the identification of regression failures, manual intervention would still be required, as indicated in Section 4.6.

5.3 Incremental Testing Approaches

Campos et al. [14] have been the first to propose a technique to incrementally test the units in a software project, leading to overall higher code coverage while reducing the time spent on test generation. They apply an evolutionary approach (i.e., EvoSuite [25]) and optimize test generation by providing more test budget to the modified portions of the code and reuse already generated test cases for seeding (i.e., they re-run all the test cases that compile). The main difference with ATUA is that they do not target the GUI testing of Apps but the unit testing of Java libraries. In our context, the reuse of existing test cases is complicated by the presence of a GUI, which is likely updated across versions and may break existing test sequences.

5.4 Testing Based on Information Retrieval

In the software testing literature, *information retrieval* techniques applied to the processing of program files have been integrated into fault localization [73, 80], test case prioritization [59], and test input selection approaches [54, 66]. Concerning test input generation, TestMiner relies on information retrieval to select from a large corpus of existing test cases the input values to use in newly generated tests cases, which differs from our purpose [66]. Poster applies a similar approach to derive sequences of Inputs for App testing [54]; different from ATUA, it relies on existing test suites developed for similar Apps.

5.5 Test Oracle Automation

Like all the state-of-the-art approaches for App testing [41, 57], ATUA does not address the *oracle problem* [11]. Oracle automation is part of our future work; however, we have presented in Section 4 two solutions to alleviate the oracle problem in the context of App updates. One solution is the automated detection of functional regression faults based on the identification of unexpected changes to outputs across different software versions [26, 37, 51, 60]. The other solution consists in relying on crowdsourcing, a popular solution adopted by industry to reduce the costs of manual GUI testing for Apps [64, 76]. Related work has shown that it is feasible for crowd workers to identify errors after visualizing the inputs and outputs of the functions under test [49]; in the App context, crowd oracles may lower testing costs while test coverage is addressed by test automation. In addition, ATUA could be integrated with approaches that automatically generate in-program logical assertions [34] or solutions relying on system-independent GUI oracles [81].

5.6 Summary

From the above, we can see that existing work lacks a model-based approach combining adaptable state abstraction functions with information retrieved from static program analysis. In ATUA, this is done to maximize the coverage of updated methods while minimizing the number of inputs required for testing. The latter is necessary to make the verification of Apps results feasible. Indeed, fully automated test oracles are currently not an option, and human effort is necessary to identify both regressions and failures in new features. Regression testing approaches, which typically target test case selection and prioritization, are of limited applicability in this context, when test suites covering large portions of the Apps code are not available. ATUA leverages incremental testing to effectively invest the test budget and maximize the coverage of updated methods.

6 CONCLUSION

State-of-the-art App testing techniques are affected by two limitations: limited effectiveness (i.e., low code coverage) and absence of automated oracles. To address the first limitation, given the high release frequency of Apps, we propose a solution (objective O1) to effectively focus the test budget on updated (i.e., modified and new) methods. In other words, within practical test execution time, we aim to maximize the coverage of updated methods and their instructions. To address the second limitation, we aim (objective O2) to generate a significantly reduced set of test inputs, compared to state-of-art approaches, thus proportionally saving the corresponding human effort required to visualize test outputs or correct test scripts.

To achieve the two objectives above, we developed ATUA, an automated App testing technique that integrates multiple analysis strategies. To achieve O1, it combines static analysis, to determine the inputs that execute updated features, and random exploration, to overcome the limitations of static analysis. To achieve O2, it relies on dynamically refined state abstraction functions, to determine when distinct inputs lead to a same program state, and relies on information retrieval techniques, to identify dependencies among App features.

We performed an empirical evaluation where we compared ATUA with state-of-the-art approaches implementing testing strategies based on dynamically derived models (DM2), random exploration (Monkey), and dynamic state abstraction (APE). For our experiments, we considered practical execution time budgets of one and five hours, corresponding, respectively, to approximate time constraints in the context of continuous integration and overnight testing. Concerning human effort (objective O2), ATUA is the approach that generates the smallest set of inputs with the highest coverage per input. ATUA, on average across subject Apps, saves around 32.6% of the effort, compared to the second-best approach (DM2). Further, it exercises 38.5% more instructions than DM2 per input. Differences with APE and Monkey are much larger. Concerning effectiveness within time budget (objective O1), on average, ATUA automatically exercises up to 70% of updated methods and 60% of instructions belonging to updated methods, 6% more than the second-best approach (i.e., DM2). These results show that the analysis strategies integrated in ATUA can drive testing towards an efficient use of the test budget (execution time and effort), thus providing clear benefits when upgrading and testing an App.

APPENDIX

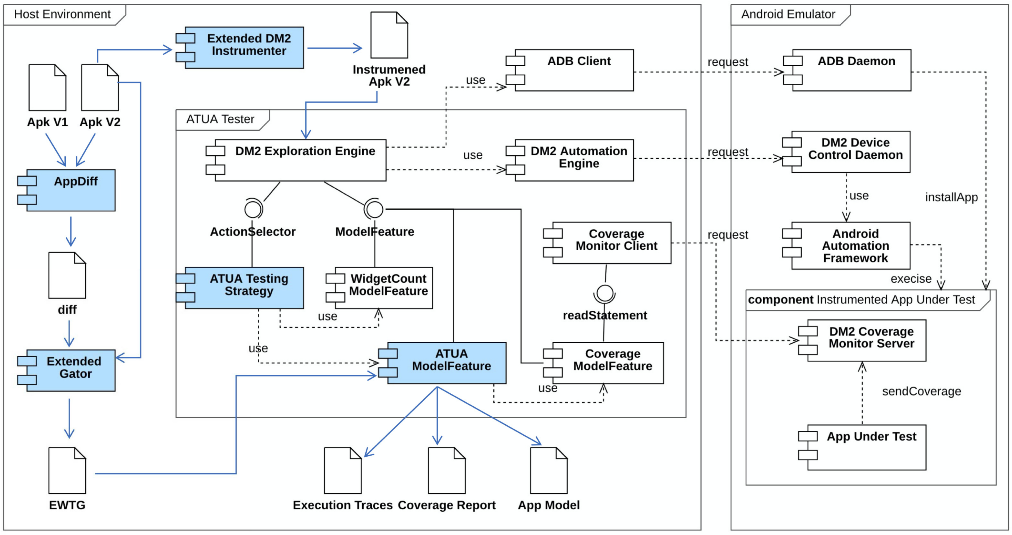
A ATUA TOOLSET

The ATUA Toolset includes four main components: *AppDiff*, which identifies the updated methods for the App under test; *Extended Gator*, which generates the EWTG part of the AppModel; *Extended DM2 Instrumenter*, which instruments the App under test; and *ATUA Tester*, which implements the ATUA testing algorithm. The UML component diagram in Figure 21 shows the ATUA Toolset.

The features of AppDiff and Extended Gator have already been presented in Sections 3.2 and 3.3, respectively. In this section, we focus on the description of the Extended DM2 Instrumenter and the ATUA Tester.

ATUA Tester has been implemented as an extension of DM2. DM2 consists of six components (i.e., DM2 Instrumenter, DM2 Exploration Engine, DM2 Automation Engine, Coverage Monitor Client, Coverage Feature, and Widget Counting Model Feature) that are executed on the host environment and two components (i.e., DM2 Control Device Daemon and DM2 Coverage Monitor Server) that are deployed on the Android emulator running the App under test. The DM2 components are part of ATUA Tester, which automatically deploys and executes them transparently from the end-user. ATUA Tester integrates two additional components that implement the ATUA algorithm (i.e., ATUA Testing Strategy and ATUA Model Feature). The integration between ATUA Tester components and DM2 is performed through the interfaces provided by DM2 (i.e., ModelFeature and ActionSelector). ATUA Tester and DM2 rely on three additional components provided by the Android development environment: the ADB Client, the ADB Daemon, and the Android Automation Framework.

The *Extended DM2 Instrumenter* is used before testing to create an instrumented version of the App under test that integrates the functions required to collect code coverage. We have extended the *DM2 Instrumenter* to collect method coverage information in addition to instruction coverage.



Legend: White UML component symbols point to third-party, reused components. Blue UML component symbols highlight components developed from scratch or extended to support ATUA's features.

Fig. 21. Overview of the ATUA toolset.

Method coverage is used by ATUA to quickly determine at runtime which methods have been covered.

At runtime, during testing, the *DM2 Exploration Engine* acts as a controller that queries the *ATUA Strategy* component, which implements the DM2 *ActionSelector* interface, used by DM2 to select the next Action to trigger during testing. The *ATUA Strategy* component implements the ATUA's testing algorithm. The DM2 Exploration Engine relies on the *ADB Client* installed on the host to set up the Android emulator and deploy the App under test. The interaction with the App under test is managed by the *DM2 Automation Engine*, which sends commands to the *DM2 Device Control Daemon* installed on the Android emulator. The *DM2 Device Control Daemon* employs the *Android Automation Framework* to execute the requested Action on the App under test and to derive the GUITree for the active Window. After triggering an Action, the *DM2 Automation Engine* receives the current GUITree, a screenshot of the Android emulator GUI, and some additional information (e.g., exception trace from Logcat) from the *DM2 Device Control Daemon*. It then derives the GUITreeTransition performed on the GSTG and sends this information to all the registered *Model Features*, including the Widget Counting Model Feature, the DM2's Coverage Feature, and the ATUA App Model. The *Widget Counting Model Feature* calculates the frequency of Actions on GUI widgets, used to drive ATUA's random exploration. The *DM2 Coverage Feature* is responsible for tracking code coverage during testing. It associates to each Action the set of instructions covered when executing the Action; code coverage is provided by the *Coverage Monitor Server* instrumented by DM2 within the App under test. The *ATUA Model Feature* updates the App model consistently with the description provided in Section 3.1; for example, it implements the ATUA state abstraction mechanism. The *ATUA Model Feature* is queried by the *ATUA Strategy* to determine the Actions to trigger (e.g., to identify the shortest sequence of Actions required to reach a TargetWindow). The *ATUA Model Feature* relies on the *DM2 Coverage Feature* to acquire the coverage data and associate them with the App Model. At the end of the testing, the ATUA

Model Feature generates ATUA's outputs (i.e., coverage report, execution traces, and the final App model).

ACKNOWLEDGMENTS

Authors would like to thank Erik Derr for the initial implementation of AppDiff.

REFERENCES

[1] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. 2017. A general framework for comparing automatic testing techniques of Android mobile apps. *J. Syst. Softw.* 125 (2017), 322–343. DOI: https://doi.org/10.1016/j.jss.2016.12.017

[2] Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. 2019. Combining automated GUI exploration of Android apps with capture and replay through machine learning. *Inf. Softw. Technol.* 105 (2019), 95–116. DOI: https://doi.org/10.1016/j.infsof.2018.08.007

[3] Android.com. 2020. Intent Resolution. Retrieved from https://developer.android.com/reference/android/content/Intent.

[4] Android.com. 2020. Logcat Command Line Tool. Retrieved from https://developer.android.com/studio/command-line/logcat.

[5] Android.com. 2020. Monkey - Android ui/application Exerciser. Retrieved from http://developer.android.com/tools/help/monkey.html.

[6] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM Press, New York, New York, 1. DOI: https://doi.org/10.1145/1985793.1985795

[7] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test., Verif. Reliab.* 24, 3 (2014), 219–250. DOI: https://doi.org/10.1002/stvr.1486

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. DOI: https://doi.org/10.1145/2666356.2594299

[9] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. Association for Computing Machinery, New York, NY, 238–249. DOI: https://doi.org/10.1145/2970276.2970313

[10] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan (Eds.). *Communications in Computer and Information Science,* Vol. 367. Springer International Publishing, 3–20. DOI: https://doi.org/10.1007/978-3-319-04519-1_1

[11] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Trans. Softw. Eng.* 41, 5 (2015), 507–525.

[12] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: A platform for Android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. ACM, New York, NY, 916–919. DOI: https://doi.org/10.1145/3238147.3240479

[13] Paolo Calciati, Konstantin Kuznetsov, Xue Bai, and Alessandra Gorla. 2018. What did really change with the new release of the app? In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*. Association for Computing Machinery, New York, NY, 142–152. DOI: https://doi.org/10.1145/3196398.3196449

[14] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. Association for Computing Machinery, New York, NY, 55–66. DOI: https://doi.org/10.1145/2642937.2643002

[15] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. 2018. DetReduce: Minimizing Android GUI test suites for regression testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. Association for Computing Machinery, New York, NY, 445–455. DOI: https://doi.org/10.1145/3180155.3180173

[16] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet? (E). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE Computer Society, Washington, DC, 429–440. DOI: https://doi.org/10.1109/ASE.2015.89

[17] Deloitte. [n.d.]. 2018 Global Mobile Consumer Survey: US Edition. Retrieved from https://www2.deloitte.com/content/dam/Deloitte/us/Documents/technology-media-telecommunications/us-tmt-global-mobile-consumer-survey-exec-summary-2018.pdf.

[18] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the 24th ACM Conference on Computer and Communication Security (CCS'17)*. ACM.

[19] Sergio Di Martino, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. 2020. Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing. *Softw. Test., Verif. Reliab.* (2020), 1–27. DOI: https://doi.org/10.1002/stvr.1754

[20] Quan Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. 2016. Regression test selection for Android applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)*. Association for Computing Machinery, New York, NY, 27–28. DOI: https://doi.org/10.1145/2897073.2897127

[21] Quan Chau Dong Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. 2016. Redroid: A regression test selection approach for Android applications. In *Procedings of the 28th International Conference on Software Engineering and Knowledge Engineering (SEKE'16)*. 486–491. DOI: https://doi.org/10.18293/SEKE2016-223

[22] Daniel Domínguez-Álvarez and Alessandra Gorla. 2019. Release practices for IOS and Android apps. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics (WAMA'19)*. Association for Computing Machinery, New York, NY, 15–18. DOI: https://doi.org/10.1145/3340496.3342762

[23] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. Association for Computing Machinery, New York, NY, 481–492. DOI: https://doi.org/10.1145/3377811.3380402

[24] Robert Feldt and Ana Magazinius. 2010. Validity threats in empirical software engineering research—An initial survey. In *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*. 374–379.

[25] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. Association for Computing Machinery, New York, NY, 416–419. DOI: https://doi.org/10.1145/2025113.2025179

[26] Z. Gao, C. Fang, and A. M. Memon. 2015. Pushing the limits on automation in GUI regression testing. In *Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE'15)*. 565–575. DOI: https://doi.org/10.1109/ISSRE.2015.7381848

[27] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2017. An exploratory study of field failures. In *Proceedings of the IEEE 28th International Symposium on Software Reliability Engineering (ISSRE'17)*. 67–77. DOI: https://doi.org/10.1109/ISSRE.2017.10

[28] Tianxiao Gu. [n.d.]. MiniTracing, APE Coverage Tool. Retrieved from http://gutianxiao.com/ape/install-mini-tracing.

[29] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. IEEE Press, 269–280. DOI: https://doi.org/10.1109/ICSE.2019.00042

[30] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. Association for Computing Machinery, New York, NY, 751–762. DOI: https://doi.org/10.1145/2950290.2950294

[31] Google Inc. 2020. Animator | Android Developers. Retrieved from https://developer.android.com/reference/kotlin/android/animation/Animator.

[32] CNRS INRIA. [n.d.]. Grid5000 Infrastructure. Retrieved from https://www.grid5000.fr.

[33] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware test-suite minimization for android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 425–436. DOI: https://doi.org/10.1145/2931037.2931067

[34] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. Association for Computing Machinery, New York, NY, 247–258. DOI: https://doi.org/10.1145/2931037.2931062

[35] Jamendo. 2020. Music Streaming App. Retrieved from https://www.jamendo.com/.

[36] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. 560–564. DOI: https://doi.org/10.1109/SANER.2015.7081877

[37] Bogdan Korel and Ali M. Al-Yami. 1998. Automated regression test generation. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98)*. Association for Computing Machinery, New York, NY, 143–152. DOI: https://doi.org/10.1145/271771.271803

[38] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of Android applications. In *Proceedings of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST'18)*. IEEE, 105–115. DOI: https://doi.org/10.1109/ICST.2018.00020

[39] K. Kuznetsov, V. Avdiienko, A. Gorla, and A. Zeller. 2018. Analyzing the user interface of Android apps. In *Proceedings of the IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft'18)*. 84–87.

[40] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li. 2017. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*. 161–171.

[41] M. Linares-Vasquez, K. Moran, and D. Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated Mobile app testing. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 399–410. DOI: https://doi.org/10.1109/ICSME.2017.27

[42] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[43] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, New York, NY, 94–105. DOI: https://doi.org/10.1145/2931037.2931054

[44] Stuart Mcilroy, Nasir Ali, and Ahmed E. Hassan. 2016. Fresh apps: An empirical study of frequently updated Mobile apps in the Google play store. *Empir. Softw. Eng.* 21, 3 (June 2016), 1346–1370. DOI: https://doi.org/10.1007/s10664-015-9388-2

[45] Chanh Duc Ngo, Fabrizio Pastore, and Lionel Briand. 2020. ATUA toolset and replicability package. Retrieved on 30 Nov, 2021 from https://github.com/SNTSVV/ATUA/.

[46] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164. DOI: https://doi.org/10.1145/3395363.3397354

[47] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li. 2019. GUI-guided repair of Mobile test scripts. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE'19)*. 326–327.

[48] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2020. GUI-guided test script repair for Mobile apps. *IEEE Trans. Softw. Eng.* 5589, c (2020), 1–1. DOI: https://doi.org/10.1109/tse.2020.3007664

[49] F. Pastore, L. Mariani, and G. Fraser. 2013. CrowdOracles: Can the crowd solve the oracle problem? In *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation*. 342–351.

[50] Fabrizio Pastore, Leonardo Mariani, Alberto Goffi, Manuel Oriol, and Michael Wahler. 2012. Dynamic analysis of upgrades in C/C++ software. In *Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE'12)*. IEEE Computer Society, 91–100. DOI: https://doi.org/10.1109/ISSRE.2012.9

[51] Fabrizio Pastore, Leonardo Mariani, Antti E. J. Hyvärinen, Grigory Fedyukovich, Natasha Sharygina, Stephan Sehestedt, and Ali Muhammad. 2014. Verification-aided regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. Association for Computing Machinery, New York, NY, 37–48. DOI: https://doi.org/10.1145/2610384.2610387

[52] Bill Phillips and Brian Hardy. 2013. *Android Programming: The Big Nerd Ranch Guide* (1st ed.). Big Nerd Ranch.

[53] Paul Ralph and Ewan Tempero. 2018. Construct validity in software engineering research and software metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering*. ACM, New York, NY, 13–23. DOI: https://doi.org/10.1145/3210459.3210461

[54] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Efficient GUI test generation by learning from tests of other apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE'18)*. Association for Computing Machinery, New York, NY, 370–371. DOI: https://doi.org/10.1145/3183440.3195014

[55] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (Apr. 1997), 173–210. DOI: https://doi.org/10.1145/248233.248262

[56] Atanas Rountev and Dacong Yan. 2014. Static reference analysis for GUI objects in Android software. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. Association for Computing Machinery, New York, NY, 143–153. DOI: https://doi.org/10.1145/2581122.2544159

[57] K. Rubinov and L. Baresi. 2018. What are we missing when testing our Android apps? *Computer* 51, 4 (Apr. 2018), 60–68. DOI: https://doi.org/10.1109/MC.2018.2141024

[58] Barbara G. Ryder and Frank Tip. 2001. Change impact analysis for object-oriented programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. Association for Computing Machinery, New York, NY, 46–53. DOI: https://doi.org/10.1145/379605.379661

[59] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. IEEE Press, 268–279.

[60] S. Shamshiri, G. Fraser, P. McMinn, and A. Orso. 2013. Search-based propagation of regression faults in automated regression testing. In *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation Workshops*. 396–399. DOI: https://doi.org/10.1109/ICSTW.2013.51

[61] Aman Sharma and Rupesh Nasre. 2019. QADroid: Regression event selection for Android applications. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. Association for Computing Machinery, New York, NY, 66–77. DOI: https://doi.org/10.1145/3293882.3330550

[62] Gwowen Shieh, Show Li Jan, and Ronald H. Randles. 2006. On power and sample size determinations for the Wilcoxon-Mann-Whitney test. *J. Nonparam. Statist.* 18, 1 (2006), 33–43. https://doi.org/10.1080/10485250500473099

[63] F. Song, Z. Xu, and F. Xu. 2017. An XPath-based approach to reusing test scripts for Android applications. In *Proceedings of the 14th Web Information Systems and Applications Conference (WISA'17)*. 143–148.

[64] STH blog editors. [n.d.]. 10 Most Popular Crowdsourced Testing Companies in 2020. Retrieved from https://www.softwaretestinghelp.com/crowdsourced-testing-companies/.

[65] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 245–256. DOI: https://doi.org/10.1145/3106237.3106298

[66] L. D. Toffola, C. Staicu, and M. Pradel. 2017. Saying "Hi!" is not enough: Mining inputs for effective test generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. 44–49. DOI: https://doi.org/10.1109/ASE.2017.8115617

[67] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: A systematic mapping study. *Softw. Qual. J.* 27, 1 (2019), 149–201. DOI: https://doi.org/10.1007/s11219-018-9418-6

[68] András Vargha and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educ. Behav. Statist.* 25, 2 (2000), 101–132. DOI: https://doi.org/10.3102/10769986025002101

[69] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: Generating high-quality test inputs for android apps via use case combinations. In *Proceedings of the International Conference on Software Engineering*. 469–480. DOI: https://doi.org/10.1145/3377811.3380382

[70] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. ACM, New York, NY, 738–748. DOI: https://doi.org/10.1145/3238147.3240465

[71] Yan Wang, Hailong Zhang, and Atanas Rountev. 2016. On the unsoundness of static analysis for Android GUIs. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State of the Art in Program Analysis (SOAP'16)*. Association for Computing Machinery, New York, NY, 18–23. DOI: https://doi.org/10.1145/2931021.2931026

[72] Christopher John Cornish Hellaby Watkins. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation. King's College, Cambridge, UK.

[73] M. Wen, R. Wu, and S. Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 262–273.

[74] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Vol. 9783642290. https://doi.org/10.1007/978-3-642-29044-2

[75] Haowei Wu, Hailong Zhang, Yan Wang, and Atanas Rountev. 2019. Sentinel: Generating GUI tests for sensor leaks in Android and Android wear apps. *Softw. Qual. J.* (2019). DOI: https://doi.org/10.1007/s11219-019-09484-z

[76] Minzhi Yan, Hailong Sun, and Xudong Liu. 2014. ITest: Testing software with Mobile crowdsourcing. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies (CrowdSoft'14)*. Association for Computing Machinery, New York, NY, 19–24. DOI: https://doi.org/10.1145/2666539.2666569

[77] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Int. J. Autom. Softw. Eng.* 25, 4 (Dec. 2018), 833–873.

[78] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static window transition graphs for Android. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 658–668.

[79] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of Mobile applications. In *Fundamental Approaches to Software Engineering*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer Berlin, 250–265.

[80] K. C. Youm, J. Ahn, J. Kim, and E. Lee. 2015. Bug localization based on code change histories and bug reports. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC'15)*. 190–197.

[81] R. N. Zaeem, M. R. Prasad, and S. Khurshid. 2014. Automated generation of oracles for testing user-interaction features of Mobile apps. In *Proceedings of the IEEE 7th International Conference on Software Testing, Verification and Validation*. 183–192. DOI: https://doi.org/10.1109/ICST.2014.31

Received November 2020; revised September 2021; accepted November 2021