ATUA: An Update-Driven App Testing Tool

Chanh-Duc Ngo
SnT Centre, University of Luxembourg
Luxembourg, Luxembourg
chanh-duc.ngo@uni.lu

Fabrizio Pastore
SnT Centre, University of Luxembourg
Luxembourg, Luxembourg
fabrizio.pastore@uni.lu

Lionel C. Briand
SnT Centre, University of Luxembourg, Luxembourg
School of EECS, University of Ottawa, Canada
lionel.briand@uni.lu

ABSTRACT

App testing tools tend to generate thousand test inputs; they help engineers identify crashing conditions but not functional failures. Indeed, detecting functional failures requires the visual inspection of App outputs, which is infeasible for thousands of inputs.

Existing App testing tools ignore that most of the Apps are frequently updated and engineers are mainly interested in testing the updated functionalities; indeed, automated regression test cases can be used otherwise.

We present ATUA, an open source tool targeting Android Apps. It achieves high coverage of the updated App code with a small number of test inputs, thus alleviating the test oracle problem (less outputs to inspect). It implements 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.

Our empirical evaluation, conducted with nine popular Android Apps (72 versions), has shown that ATUA, compared to state-of-the-art approaches, achieves higher code coverage while producing fewer outputs to be manually inspected.

A demo video is available at https://youtu.be/RqQ1z_Nkaqo.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Android Testing, Regression Testing, Upgrade Testing

ACM Reference Format:
Chanh-Duc Ngo, Fabrizio Pastore, and Lionel C. Briand. 2022. ATUA: An Update-Driven App Testing Tool. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3533767.3543293

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3543293

1 INTRODUCTION

App testing tools can be categorized according to the strategy adopted to generate test inputs [10]. The most common ones are random, model-based, and evolutionary [10]. Unfortunately, they all typically exercise about half of the methods implemented by commercial apps [11]. Further, they cannot discover functional failures beyond crashes; indeed, the detection of functional failures requires the manual verification of the App outputs to determine if they match requirement specifications. However, such manual verification is rendered infeasible by the large number of inputs exercised by App testing tools [3]. Limiting the number of test inputs is therefore important to minimize human effort.

Further, existing tools do not support prioritizing the testing of updated (i.e., modified or newly introduced) features, which is particularly crucial given that Apps are frequently released [4]. Exercising all the features of an App in each release is enormously wasteful; indeed, the test budget is used to trigger all the App features, including those that are not updated. Consequently, updated features may remain uncovered. If the available test cases do not exercise all updated App features, then the selection of regression test cases is of limited usefulness [3, 9].

We present ATUA, a tool that implements *Automated Testing of Updated Apps*, our App testing technique [6]. ATUA achieves the following objectives: (1) maximize the number of updated methods and instructions that are automatically exercised within practical test execution time, and (2) generate a significantly reduced set of inputs, compared to state-of-art approaches, thus facilitating manual inspection and the detection of functional faults.

ATUA combines static and dynamic program analysis to cost-effectively select the inputs that exercise updated methods, our test targets. Also, to deal with the complexity of Apps, it relies on three incremental testing phases, which focus on objectives of increasing complexity. In the first phase, it exercises all the features that may trigger updated methods. In the second phase, it exercises updated features with diverse input values, to maximize coverage in the presence of data dependencies. In the third phase, it exercises features related to the updated ones to satisfy state dependencies. ATUA implements a model-based approach that integrates *dynamically-refined state abstraction functions* and relies on complementary testing strategies: (1) coverage of the *model structure*, (2) coverage of the *App code*, (3) *random* exploration, and (4) coverage of *dependencies* among App windows.

An empirical evaluation conducted with nine popular Apps has shown that ATUA, compared to state-of-the-art tools (i.e., DM2 [2], APE [5], and Monkey [1]), leads to reduced test costs. It generates less than 70%, 4%, and 2% of the inputs generated by DM2, APE, and Monkey, respectively. Further, for the same test execution budget (e.g., 1 hour test execution time), it improves the method and instruction coverage achieved by the second best tool by at least 10%.

In the following we provide an overview of the ATUA approach (Section 2), describe ATUA's architecture (Section 4), and summarize our empirical evaluation (Section 5).

2 THE ATUA APPROACH

We aim to exercise *updated features* automatically; more precisely, we focus on features that are implemented or repaired either by introducing new methods or by modifying existing ones. We call these methods *target methods* since they are our test targets.

The testing activity performed by ATUA is driven by an App model that 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). They are finite state machines capturing how input values trigger changes in the state of the App under test. The *EWTG* is extracted through static analysis [8]; it models the sequences of windows being visualized after triggering specific inputs (Events or Intents). For every input, the EWTG traces the list of target methods 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 position of a button) that might be triggered by an action performed 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 and aim at eliminating non-determinism. The DSTG plays a critical role in optimizing the test budget and identifying a reduced set of input events necessary to reach a specific Window from another one.

To test an App, ATUA identifies the shortest sequence of inputs necessary to reach a target Window (i.e., a Window where some inputs may trigger the execution of target methods). It relies on a breadth-first traversal that considers both Window transitions in the EWTG and state transitions in the DSTG. Once in a target Window, ATUA triggers inputs that may exercise target methods.

During testing, ATUA identifies abstract states through *abstraction functions* that are automatically generated for each Window. Abstraction functions rely on a predefined set of *reducers*, i.e., functions that extract the value of a widget property (e.g., the value of the resource ID) [5] . For each Widget, ATUA keeps track of the applied reducers and the values returned by each in a map called AttributeValuationMap. Two abstract states differ when at least one value differs across their respective AttributeValuationMaps.

To minimize non-determinism in the DSTG, during testing, when an action does not bring the App into the expected abstract state, ATUA refines the abstraction function for the Window in which the action has been triggered. Refinement is performed by increasing the number of reducers used.

Testing is performed in three phases. Phase 1 aims at maximizing the number of target transitions being exercised, which is obtained by triggering all the target inputs of every target Window. Phase 2 aims at increasing testing of the less exercised target transitions, which is obtained by exercising, with different values (e.g., numbers, empty strings, and regulars words for text areas), inputs that may reach target methods not fully covered. Phase 3 aims at exercising target Windows that depend on specific App states set in related Windows; it consists of relying on information retrieval techniques to determine what are the Windows that should be exercised in combination with target Windows.

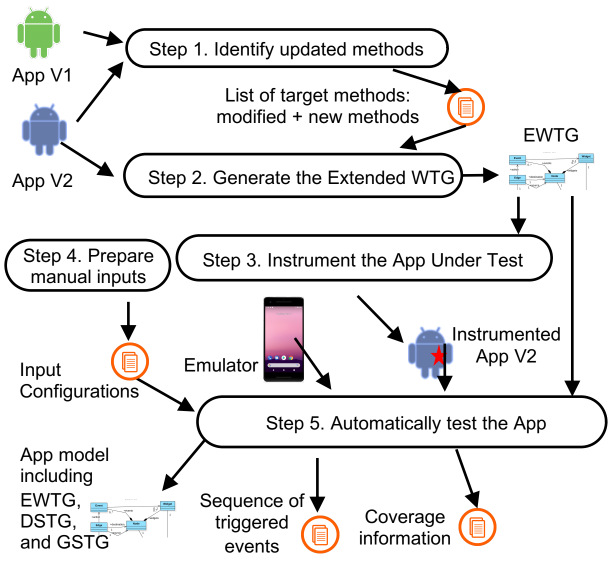

App V1
App V2
Step 1. Identify updated methods
List of target methods: modified + new methods
Step 2. Generate the Extended WTG
EWTG
Step 4. Prepare manual inputs
Step 3. Instrument the App Under Test
Input Configurations
Emulator
Instrumented App V2
Step 5. Automatically test the App
App model including EWTG, DSTG, and GSTG
Sequence of triggered events
Coverage information

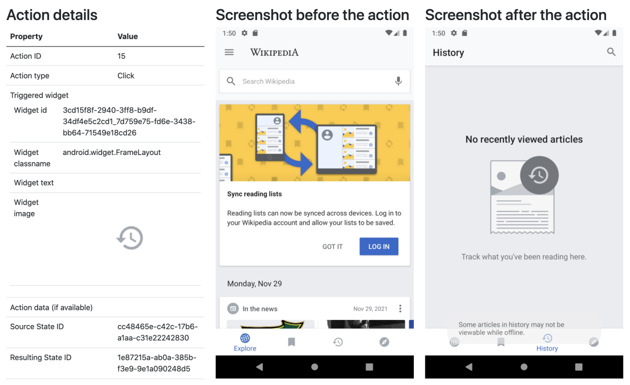
Figure 1: Overview of the ATUA process to test App updates.

Part of the test budget is spent to alleviate the limitations of static analysis through dynamic analysis. More precisely, ATUA needs to resort to random exploration when Window transitions are not feasible (e.g., it is necessary to find the abstract state in which they are enabled), when Windows are unreachable (e.g., if static analysis does not detect any Window transition reaching a specific Window) or when they are incompletely processed by static analysis (e.g., they reach a target method but not through an input handler).

Figure 1 provides an overview of the process implemented by ATUA. In Step 1, ATUA compares the previous and updated versions of the App under test to identify the target methods. In Step 2, ATUA relies on an extended version of Gator, to generate the EWTG. In Step 3, ATUA generates an instrumented version of the App that traces code coverage. In Step 4, engineers manually specify input values that are unlikely to be generated automatically (e.g., login credentials). In Step 5, ATUA exercises the App under test following the test procedure described above.

The output of ATUA (i.e., Step 5) is an App model for the updated App version. Also, ATUA generates a report with a set of triples <GUI screenshot, target action, GUI screenshot> reporting for every target action (i.e., an action that triggers the execution of a target method) triggered by ATUA the screenshot before and after the execution of the action[1]. An example is shown in Figure 2. To prevent wasting engineering time, ATUA reports only actions that increase code coverage (i.e., exercising instructions not tested yet).

[1]Such report is a new contribution with respect to our previous work [6].


Action details

Property	Value
Action ID	15
Action type	Click
Triggered widget	
Widget id	3cd15f8f-2940-3ff8-b9df-34df4e5c2cd1_7d759e75-fd6e-3438-bb64-71549e18cd26
Widget classname	android.widget.FrameLayout
Widget text	
Widget image	
Action data (if available)	
Source State ID	cc48465e-c42c-17b6-a1aa-c31e22242830
Resulting State ID	1e87215a-ab0a-385b-f3e9-9e1a090248d5

Screenshot before the action

1:50

Wikipedia

Search Wikipedia

Sync reading lists

Reading lists can now be synced across devices. Log in to your Wikipedia account and allow your lists to be saved.

GOT IT LOG IN

Monday, Nov 29

In the news Nov 29, 2021

Explore

Screenshot after the action

1:50

History

No recently viewed articles

Track what you've been reading here.

Some articles in history may not be viewable while offline.

History

Figure 2: An action output being reported to the end-user

3 RUNNING EXAMPLE

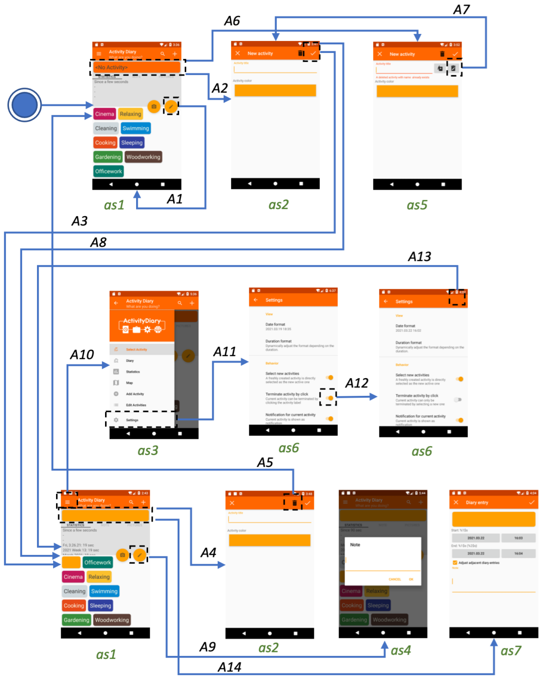
Figure 3 provides a running example for ATUA testing the Activity Diary App. In Phase 1, ATUA clicks on the edit note button (A1) and, since there is no current activity selected, partially covers the target methods (the EditNote Window is not opened). A long click on the current activity widget (A2) leads to an instance of the EditActivity Window, which is randomly explored till the activity is saved. In Phase 2, because of the incomplete coverage mentioned above, ATUA again exercises the EditNote button (A9), which pops up the EditNote Dialog thus covering the missing lines. In Phase 3, ATUA tests the feature that visualizes the details of the current activity after a click on the current activity widget. ATUA selects the SettingsActivity as a Window related to MainActivity. While exercising the SettingsActivity, it deselects the option *Terminate activity by click* (A12), which enables exercising the updated feature (A14). Phase 3 enabled ATUA to test the updated feature in a few steps, which is unlikely with state-of-the-art approaches.

4 ATUA TOOLSET ARCHITECTURE

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 diagram in Figure 4 depicts the ATUA Toolset.

ATUA Tester has been implemented as an extension of DM2; it integrates six DM2 components and two additional components that implement the ATUA algorithm (i.e., ATUA Testing Strategy and ATUA Model Feature). The integration between ATUA Tester and DM2 is based on the interfaces provided by DM2. ATUA Tester and DM2 rely on 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 to create a version of the App under test that collects method coverage information in addition to instruction coverage (i.e., provided by *DM2 Instrumenter*). Method coverage is used by ATUA to determine which methods have been covered quickly.



Legend: Arrows show the sequence of actions triggered by ATUA.

Figure 3: Running example.

During testing, the *DM2 Exploration Engine* acts as a controller that queries the *ATUA Strategy* component, which implements the ATUA's testing algorithm. After triggering an action, the *DM2 Automation Engine* requests the current GUI Tree and a screenshot of the Android GUI from the *DM2 Device Control Daemon*. It then derives the state transition performed on the GSTG and sends this information to the ATUA Model Feature, which updates the App model.

The *ATUA Model Feature* is queried by the *ATUA Strategy* to determine the actions to trigger. At the end of the testing, the ATUA Model Feature generates ATUA's outputs.

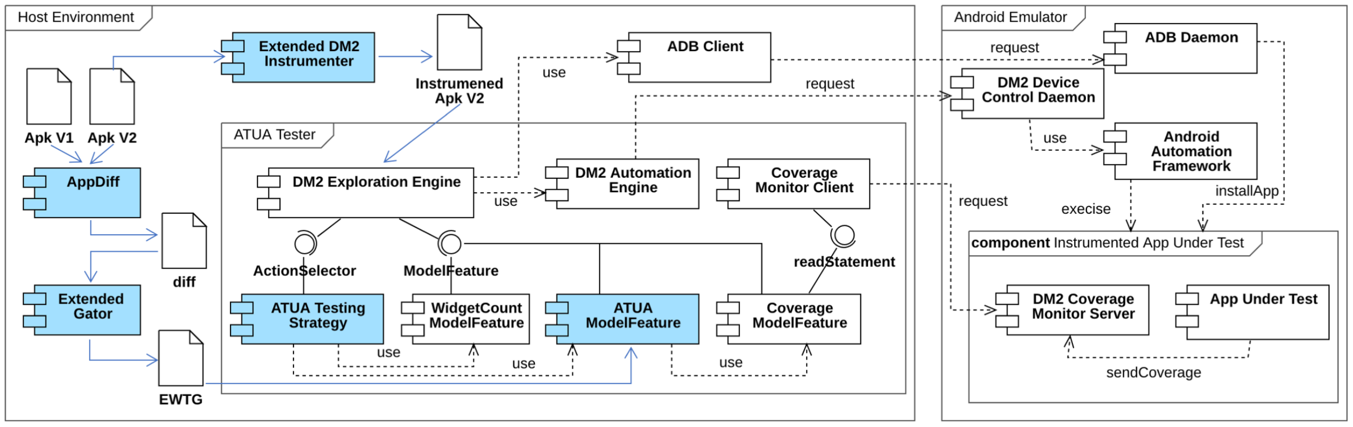
5 EMPIRICAL EVALUATION

We conducted an empirical evaluation to compare ATUA with state-of-the-art approaches (i.e., DM2, Monkey and APE) in cost-effectiveness; please refer to our journal publication for details [6].

We considered a representative set of nine Apps, including open source (i.e., Activity Diary, File Manager, Wikipedia, VLC player and Nuzzle) and proprietary ones (BBC Mobile, Citymapper, YahooWeather, Wikihow). Each App was tested with up to 10 versions. In total, our subjects include 81 App versions.

In our experiments, we considered two possible execution scenarios, with respectively test budgets of one hour and five hours. We executed each tool on each updated version 10 times. In total, we executed 5760 test sessions for a total of 17280 test execution hours. A replicability package is available online [7]. Below, we describe our research questions and summarize the results achieved.

RQ1. *Can ATUA reduce the human effort required for testing Apps, compared to state-of-the-art approaches?* We aim to determine if the



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

Figure 4: Architecture of the ATUA toolset.

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. Our results show that **ATUA significantly decreases the human effort required for verifying inputs when compared to state-of-the-art approaches**. Indeed, it generates less than 70%, 4% and 2% of the inputs generated by DM2, APE and Monkey, respectively. If engineers aim to inspect only the outputs generated when covering updated methods (a feature not supported by other tools), with ATUA they will inspect only 30 output screens (one example output appears in Figure 2) per App version, on average. Such limited number of outputs can realistically be manually inspected, thus demonstrating that ATUA significantly alleviates the oracle problem.

RQ2. *Can ATUA effectively test Apps within practical time budgets, compared to state-of-the-art approaches?* We aim to determine if ATUA performs significantly better than state-of-the-art approaches in terms of coverage of updated methods and their instructions, for a same execution time budget. Our results demonstrate that **ATUA is the approach that, on average, most effectively tests updated Apps within practical time budgets and human effort**. On average, for the same test execution budget, ATUA improves the method and instruction coverage achieved by the second-best, state-of-the-art approach by at least 10%.

RQ3. *Is there any difference in the functionalities that are automatically exercised across test automation approaches?* We aim to determine if there are differences in the inputs triggered by the different approaches that lead to a diverse and complementary set of functionalities being exercised. Our results show that **ATUA is the approach that exercised the largest number of uniquely covered methods** (i.e., not covered by other approaches). Such differences are mainly achieved thanks to the three different test phases implemented by ATUA.

6 CONCLUSION

ATUA is the first App testing tool that effectively focuses the test budget on updated methods. Such strategy enables ATUA to maximize the coverage of updated methods (a primary need for engineers) and generate a significantly reduced set of test inputs, compared to state-of-art approaches, thus proportionally saving human effort required to visualize test outputs.

Among state-of-the-art solutions, ATUA is the tool that generates the smallest set of inputs with the highest coverage per input. It automatically exercises up to 70% of updated methods and 60% of instructions belonging to updated methods, 6 percentage points more than the second best approach. On average, with ATUA, engineers should inspect 30 output screens per App. ATUA is opensource and available at https://github.com/SNTSVV/ATUA.

ACKNOWLEDGMENTS

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).

REFERENCES

[1] Android. 2022. Monkey. http://developer.android.com/tools/help/monkey.html.
[2] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: A Platform for Android Test Generation. In *ASE 2018*. ACM, New York, NY, USA, 916–919. https://doi.org/10.1145/3238147.3240479
[3] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. 2018. DetReduce: Minimizing Android GUI Test Suites for Regression Testing. In *ICSE 2018*. ACM, New York, NY, USA, 445–455. https://doi.org/10.1145/3180155.3180173
[4] Daniel Domínguez-Álvarez and Alessandra Gorla. 2019. Release Practices for IOS and Android Apps. In *WAMA 2019*. ACM, New York, NY, USA, 15–18.
[5] 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 *ICSE 2019*. IEEE Press, 269–280. https://doi.org/10.1109/ICSE.2019.00042
[6] Chanh Duc Ngo, Fabrizio Pastore, and Lionel Briand. 2021. Automated, Cost-effective, and Update-driven App Testing. *ACM TOSEM* (Dec. 2021). https://doi.org/10.1145/3502297 arXiv: 2012.02471.
[7] Chanh Duc Ngo, Fabrizio Pastore, and Lionel Briand. 2022. ATUA replicability package. https://doi.org/10.5281/zenodo.5734090.
[8] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *CGO 2014*. ACM, New York, NY, USA, 143:143–143:153.
[9] Aman Sharma and Rupesh Nasre. 2019. QADroid: Regression Event Selection for Android Applications. In *ISSTA 2019*. ACM, New York, NY, USA, 66–77.
[10] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal* 27, 1 (2019), 149–201.
[11] 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 *ASE 2018*. ACM, New York, NY, USA, 738–748.