Capture & Replay with Text-Based Reuse and Framework Agnosticism

Filipe Arruda, Augusto Sampaio and Flavia Barros
Centro de Informática
Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil
{fmca, acas, fab}@cin.ufpe.br

***Abstract*—Software systems need to be constantly tested, either to verify changes or to check conformance to requirements. The current leading approaches to automate GUI tests are coding and the use of Capture & Replay (C&R) tools. Coding is usually associated with (even if *ad hoc*) reuse strategies, but requires from the developer specialized knowledge about the adopted framework. On the other hand, even though C&R is able to promote faster automation, it raises maintainability and scalability issues in the long term due to scripts scattering and rework for each new test case, because usually there is no associated reuse strategy. In order to combine the benefits of both approaches, we propose: an abstract and framework-free representation of test actions captured during testing activities; a text-based strategy that matches a new test case with previously recorded test actions; and a C&R tool that implements these concepts in the mobile context. We developed and evaluated our strategy in the context of a partnership with Motorola Mobility, achieving a reuse ratio up to 71% with time gains similar to traditional C&R approaches when compared to coding.**

***Keywords*-test automation; capture and replay; reuse; mobile applications; natural language processing**

I. Introduction

Despite the consolidation of testing as a verification activity, it is not always feasible to complete a testing campaign due to budget and time constraints [1]. Furthermore, when we consider the context of mobile devices, which is considerably different from Web and Desktop contexts, several other aspects should be observed, such as the wide range of sensors, distinct network providers and strong hardware dependency [2]. Thus, a large number of test cases (TCs) tends to be necessary to cover all these aspects, increasing the cost of the process.

Testing tasks tend to be repetitive. In Regression testing campaigns, for instance, every new version of the software must be tested, to verify whether it behaves as expected and to detect errors that may have been introduced in the modified code [3]. The simplest regression testing strategy consists of retesting the entire software by rerunning the reference test suite. However, due to cost constraints, alternative approaches select and run only a subset of the test suite [4]. In this context, test automation is seen as a way to reduce the time spent on testing activities [5][6], by mitigating the effort to manually execute an entire test suite. To automate test execution, Capture & Replay (C&R) tools are an alternative which does not demand programming skills and can be used during a testing campaign while tests are performed manually.

DOI reference number: 10.18293/SEKE2016-228

Regression testing arguably helps to control software quality, but new TCs must be continuously created to cover new features [7]. In some software systems, new features could have a high demanding rate, and automating the TCs for these features may be time consuming: 1) coding forces developers to acquire deep knowledge of the testing framework, design patterns etc. 2) current C&R tools require testers to execute the entire TC at least once. Besides, C&R approaches also suffer from high maintenance costs due to poor reuse, as noted, for instance, in the empirical assessment conducted in [8]. These evidences, associated to the use of an ambiguous language to specify requirements, affect negatively a direct mapping from test descriptions to scripts by automation tools, demanding testers to create a structured representation to enable an automatic and efficient transformation [9]. However, practical experiences in testing have shown that forcing programmers to adopt new notations is not the best option, reinforcing the use of well-known notations and environments [10][11].

Considering this scenario, we propose an abstract, recursive, text-based and framework-free representation, named *test action*, to store information that ranges from a simple test step to a complete TC or even a test suite. We apply text-processing algorithms to match new TC descriptions in natural language to previously recorded actions, reusing them to automate new TCs. We implemented these concepts into a C&R tool: *AutoMano*. Our tool was evaluated in an industrial context of a partnership with Motorola Mobility considering two metrics: reuse ratio and time spent to automate. As an important result we report a reuse ratio of 71% and a significant reduction in the overall implementation effort.

Section 2 discusses related work. Section 3 describes our proposed representation for test actions and our approach to reuse this representation by using string proximity and synonym matching techniques. Section 4 describes the tool and Section 5 details the conducted evaluation. Finally, Section 6 brings conclusions and future work.

II. Related Work

Due to the vast literature on test automation, we focus here on approaches that are closely related to our work. UIAutomator[1] and Espresso[2] frameworks, developed by Google,

[1]http://developer.android.com/intl/en-us/tools/testing-support-library
[2]https://code.google.com/p/android-test-kit/wiki/Espresso

allow test automation through coding based on atomic actions (e.g. click.id(text1)). However, the use of these frameworks demands specialized knowledge from the developer. Moreover, the automated TCs are framework dependent, and may become outdated when they use features that have been discontinued/deprecated after a framework or system update, requiring significant code refactoring. To minimize these effects, *AutoMano* proposes an intermediate representation and a central database that can be easily queried and updated.

Automation based on the C&R approach, on the other hand, does not require prior knowledge and is very fast. However, the created test scripts are typically linear, in the sense that they do not embody alternative paths, serving only the purpose of purely playback. Note that this characteristic restricts the reuse of actions to compose new TCs – for instance, just a slightly different disposal of the GUI may fail the execution of the test, as in the RERAN tool [12].

There are also hybrid approaches that mix the C&R approach with a strategy to capture keywords from the GUI to compose test actions. The test actions are stored as a script to be reproduced later. MonkeyTalk [13] and Robotium Recorder [14] are examples of frameworks that support C&R of keywords from applications built over Android or IOS platforms. Although they are well-consolidated commercial tools, they do not link test actions with TC descriptions (which is an important link we explore to ease the process of designing and reusing TCs). Moreover, even though we focus our comparison on C&R versus coding approaches for test automation, there are also other well-known strategies such as model-based testing (an example is MobiGUITAR [15]) and GUI ripping (see, for instance, [16]).

Our work was also inspired by an empirical analysis seen in [8], which shows that the development of test suites requires more time when programmable testing approaches are adopted (between 32% and 112%) compared to C&R approaches, however it is more time-saving over successive releases, because it is easier to maintain. Although these results were obtained in a different context (web), we assume that they could be similar or even more prominent in the mobile context. This happens because no reuse strategy are applied in C&R artifacts, while coded tests benefit from design patterns such as Page Objects. In our strategy we take advantage of the fast development observed in C&R tools without suffering maintenance issues raised from this approach by applying a reuse strategy.

Concerning TC generation from natural language requirements, NAT2TEST [17] presents a strategy that maps requirements written in natural language into TCs using a formal notation for requirements specification (SCR) as an intermediate formalism. Another example is presented in [18], in which requirements must be written in a more restrictive way as a strict if-then sentence template. Although these approaches aim to simplify the generation of TCs, these works: (1) severely restrict the input to a subset of a given natural language that must obey a particular grammar; (2) do not consider the reuse of test artifacts; (3) are not applicable to C&R approaches.

III. Automation Strategy

Our automation strategy is based on C&R, with focus on improving previous approaches particularly concerning potentializing reuse. Also, the input to our strategy is a TC written in free Natural Language (NL), particularly English; as far as we are aware, currently there is no strategy that automatically translates TCs written in (an unconstrained) NL into scripts of an automation framework, exploring reuse.

As previously discussed, the development of automated TCs, via C&R, usually results in platform dependent and hard-to-maintain code. For instance, considering the automation of a TC illustrated in Figure III, a direct mapping to scripts, besides the difficulty to find a connection between these representations, also hinders reuse possibilities. One could argue that, instead of mapping a TC to a single script, modularization could be explored by assigning a script to each step and reuse them in other TCs; but this is not sufficient to represent hierarchical steps. In addition, a direct mapping also makes scripts framework-dependent.

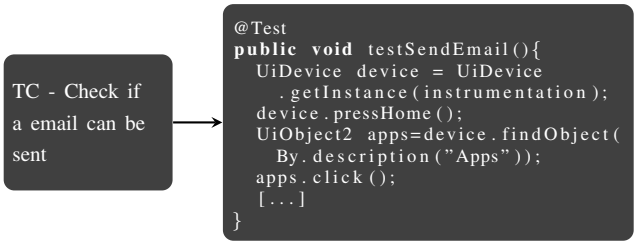

Fig. 1. Typical TC automation based on Capture & Replay

We propose the use of a middle-level representation, called *test action*, that fills the granularity gap between NL descriptions and GUI operations by supporting abstraction layers, composition and code-level interpretation. In this way, besides allowing actions to be retrieved, composed and reused by the test description itself, it is also possible to mitigate inconsistencies due to framework changes (deprecation, switched platform, business decision etc.), as detailed next.

A. Overall Architecture

In order to provide a better representation of the underlying abstractions of NL descriptions, test actions can be represented as recursive structures, inspired by the composite design pattern [19] illustrated in Figure 2 (a), supporting layers of abstractions that allow one to represent atomic operations, test steps, TCs or even test suites using the same structure.

It is worth mentioning that the high-level descriptions of atomic operations are automatically derived from screen interactions, while in composed test actions these NL descriptions are TC titles, step descriptions etc. Only atomic operations (that are predefined) are mapped to code-level scripts by using an interpreter to a specific framework (Figure 2 (b)), which can be dynamically instantiated using the factory method pattern [19]. To illustrate the framework agnosticism, we consider (in Figure 2 (b)) two frameworks: *UiAutomator* and *MonkeyTalk* with their respective interpreters.

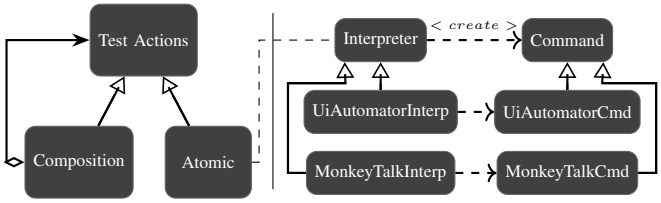

Fig. 2. Overall architecture

For instance, given the TC illustrated in Figure III to check whether an email can be sent, composed by several steps, the test action that represents this TC can now be structured as a composition of other test actions (each one representing a step), which in turn could also be composed by several screen interactions (represented as atomic test actions), as presented in Figure 3. We potentialise the reuse possibilities and provide the code script by only interpreting the atomic actions.

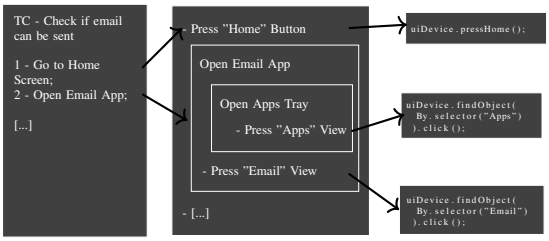

Fig. 3. Test case automation using hierarchical test actions

B. Reuse

In the beginning of an automation process, there are no previously created TCs, so there is no opportunity for reuse. However, as TCs are progressively automated, test actions are stored (typically in a database, as discussed in the next section) and can then be retrieved and reused in the automation of new TCs. Therefore, instead of capturing interactions all over again for every new TC, we employ an algorithm to match test steps written in English with the stored test actions, mitigating C&R issues regarding reuse, as noted in the empirical assessment conducted by [8]. These test actions may be organized and composed in any order to create others yet more complex ones.

The matching process is divided into three main operations: sentence tokenization; synonyms retrieval from the WordNet knowledge database [20]; and finally ranking the test actions using synonym equivalence and string proximity. The whole matching process is detailed in Algorithm 1. The function described as *CalculateSimilarity* is used to search test actions that match each step of TC description written in natural language. This function receives a test step and a description of a given test action as arguments, both written in English. Then, at lines 2 and 3, the sentences are split, also discarding some *stop words*. Then, beginning at line 5, each word of a test step (kwTS) is compared to each word of a test action (kwAD) and its synonyms retrieved from WordNet by verifying the Levenshtein distance [21]; for each successful matching (given a threshold), the similarity level is increased. This function is executed for all stored action textual descriptions to find the best similarity level. Finally, if no test action found is similar enough (for a given threshold), the user must enter or create one specific test action for the test step being processed. For instance, if we consider a given TC step: "Compose a POP mail", the matching process is illustrated in Figure 4.

Algorithm 1 Calculating similarity using a knowledge-based approach

```
1:  function CALCULATESIMILARITY(testStep, actionDescription)
2:      kwTS ← GETKEYWORDS(testStep)
3:      kwAD ← GETKEYWORDS(actionDescription)
4:      similarity ← 0
5:      for i ← 0..SIZE(kwTS) do
6:          synonyms ← GETSYNONYMS(kwTS[i])
7:          for j ← 0..SIZE(kwAD) do
8:              kwContains ← CONTAINS(synonyms[s], kwAD[j])
9:              kwProximity ← DISTANCE(synonyms[s], kwAD[j])
10:             if kwContains OR kwProximity < threshold then
11:                 similarity ← similarity + 1/(COUNT(kwTS))
12:                 break
13:             end if
14:         end for
15:     end for
16:     return similarity
17: end function
```

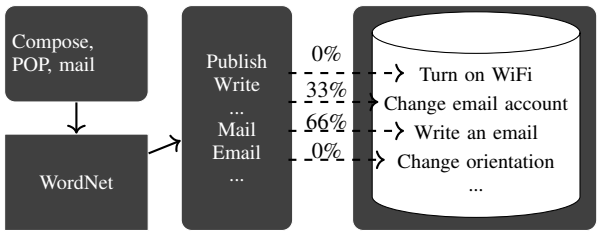

Fig. 4. Matching process

IV. Implementation

We implemented a tool, called AutoMano, which is able to capture user interactions on the phone and store them as test actions. It is worth noting that the application UI is web-based and this matching is transparent to the user. In summary, as in other C&R tools, a tester is able to automate an entire test suite without any programming skills during her common activities, reducing both time and effort to automate tests. Unlike other tools, however, it is possible to easily automate new TCs by just typing their descriptions (which potentially match actions previously recorded), favoring reuse because all actions are expressed as English sentences.

A. Capture & Replay

As observed in traditional C&R tools, AutoMano also captures and stores user inputs to reproduce them later. However, instead of capturing low-level events such as "clicking on (x,y)", our tool listens to the Android accessibility events[3], which give us high-level descriptions of what was performed on the device. In this way we mitigate screen compatibility issues (due to different screen sizes), besides giving a more legible way to present information to the user. For instance, instead of "click on point (x,y)", our tool captures "click on

[3]http://developer.android.com/intl/en-us/reference/android/view/ accessibility/AccessibilityEvent.html

button with description 'Apps'". Also, we created a custom keyboard that we install before capturing to get what the user is typing. Additionally, the user can create variables to reuse the same action in other situations, as shown in Figure 5.

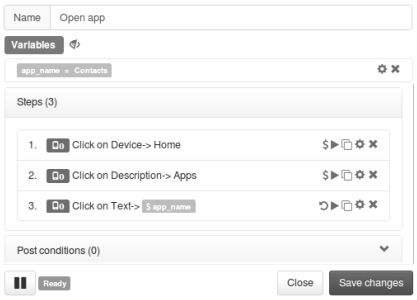

Fig. 5. Capture screen (AutoMano)

Then, after capturing a set of interactions, the user should give a representative description in English before storing them in a database (so these interactions could be retrieved by the algorithm presented in the previous section). The database used was Neo4j[4] because it enables us to make graph queries which are useful to find equivalent subgraphs and analyze transitive connections among test actions. An example of how a test action would be represented is shown in Figure 6.

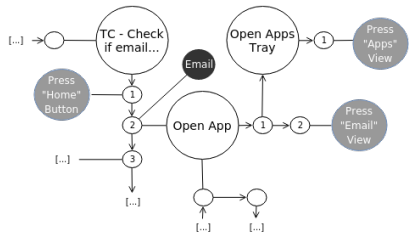

Fig. 6. How test actions are represented in Neo4j

Afterwards, to execute the test actions previously recorded, we first install an interpreter that reads test actions (JSON) and calls the corresponding framework (e.g., *UiAutomator*) methods. Then, the test action is retrieved from the database, converted to JSON and sent to device via socket communication. Subsequently, the interpreter translates each command of a test action, executes it and sends back the action results. Currently, our interpreter generates scripts for two frameworks: *UiAutomator* and a proprietary one. It is important to emphasize, however, that AutoMano can be easily tailored to use any other automation framework, since all commands are dynamically linked to scripts of a chosen framework. Finally, the results collected are shown on an HTML page to the user.

V. Evaluation

In order to evaluate the effectiveness of our approach, we conducted an experiment in which we measured the time spent in test automation by developers coding scripts and that spent by testers using AutoMano for C&R. In addition, we carried out a longitudinal analysis to investigate the evolution of the reuse ratio along a real-world automation task assigned to a tester in a project within Motorola Mobility. In this section we summarize the research questions, design choices, results and threats to validity.

A. Research Questions and Metrics

As a measurement mechanism, we adopted the GQM (Goal, Question, Metric) approach [22] to structure our evaluation in conceptual, operational and quantitative levels.

Study Main Goal: Verify whether our proposed C&R approach potentialise the reuse ratio of test actions during an automation task, while still being faster than coding.

[RQ1] Research Question 1: Is there indeed a reduction of the time spent to automate TCs via C&R with AutoMano when compared to coding?

Metric: Time spent to automate a set of TCs subtracting the execution and preparation time.

[RQ2] Research Question 2: Does our approach provide a satisfactory reuse ratio during automation?

Metric: Percentage of reused test actions: number of test steps retrieved from the database divided by the total number of test steps.

B. RQ1 - Design, Execution and Results

To obtain the metric associated with RQ1, we conducted an experiment that aimed to compare the time spent by developers to automate a given set of TCs with the time spent by testers to automate the same TCs via C&R, using AutoMano. These TCs were chosen according to the following criteria:

- They should be simple enough to mitigate the issue of different expertise levels of developers and testers.
- They should address recently added features, preferably not known by participants beforehand (to neutralize the testers prior knowledge).
- They must be real TCs, written by test designers under the same principles.

To conduct a precise computation of the time aspect, we prepared some guidelines and rules to the participants:

- Immediately before starting the automation activity, start the chronometer;
- Before asking for help, pause the chronometer and only resume it when the issue is solved;
- Pause the chronometer after finishing the automation of each TC, since the execution time to check if there are no errors should not be taken into account.

Regarding the selection of the participants, we chose 10 developers from an automation team working for Motorola Mobility whose experience ranged from preliminary to substantial. The project managers also assigned 10 testers with varied experience levels, who were trained for about 20 minutes on how to use the AutoMano tool. However, we ran the experiment with 7 available developers and 10 testers.

To ensure a proper execution, two assistants were assigned to supervise the experiment, enforcing the guidelines and

[4]http://neo4j.com/

assuring that the TCs were correctly automated. Whenever a problem was detected, the chronometer was stopped until the issue was fixed. Participants were not allowed to use functions or actions created prior to the current evaluation process. This way, TCs had to be automated from scratch. Additionally, it is worth mentioning that all participants used the same device and software version, and developers were instructed to use the same automation framework.

We protected the identity of the participants, to avoid any kind of retaliation due to performance indicators. As such, we assured to only share consolidated results.

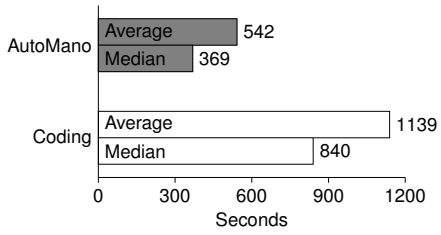

Fig. 7. Results: Comparing time to automate (AutoMano x Coding)

As shown in Figure 7, coding required an average of 101% more time to automate when compared to AutoMano, which actually is in accordance to the results achieved by Leotta et. al. (2013) [8], besides providing us a concrete evidence that AutoMano is faster than coding under these circumstances. To reach this outcome, we disregarded the values with more than 1.5 standard deviations from the mean, which excluded the highest automation time from both groups.

C. RQ2 - Design, Execution and Results

Concerning the RQ2, we prepared some artifacts to analyze the reuse ratio of test actions during a real task assignment in a Motorola Mobility project. For 3 weeks, we observed a novice tester using AutoMano to automate a set of TCs. Before we started to observe the tester, he was trained for 1 week on minor tasks using AutoMano, so that he could grasp the basic concepts related to testing activities, automation and the AutoMano itself. The aim was to provide a fairly meaningful analysis. The tester was asked to register each test step automated with AutoMano, informing if it was necessary to record the interactions or if there was a correspondent action already recorded in the database.

To track the reuse ratio evolution, we collected the percentage of reused actions over a time period. Then we observed how many actions were reused relative to the number of TCs automated. Figure 8 shows how reuse improved along the automation task. In general, as the number of TCs increased, the reuse ratio increased as well, reaching 71% with 31 TCs.

D. Threats to Validity

We discuss here the threats to validity of our evaluation.

Conclusion Validity: This work and evaluation were based on the assumption that it is reasonable to compare the time to automate TCs between two different groups: developers and testers. We could have designed an experiment with only developers, but it would not have any practical or meaningful results since we seek alternatives to ease the burden of developers to automate and maintain a large set of TCs (by assigning them to testers, with no programming background, that could automate them even faster). Concerning the reuse strategy, we assumed that TCs have descriptions written in English and could be broken down into steps.

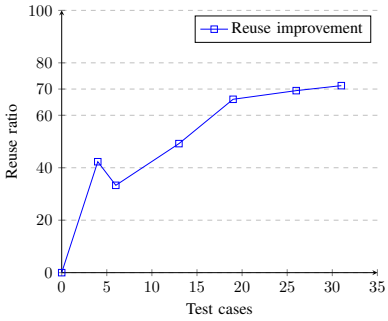

Fig. 8. Longitudinal analysis: Reuse ratio

Internal Validity: Regarding the reuse ratio, the tester may have chosen only a subset of similar TCs, boosting the percentage of reused actions. If TCs from different products/projects are chosen, the reuse ratio might be distinct. However, we believe that as the number of TCs increase, we would notice a similar result. Concerning the experiment (RQ1), some participants could have measured a wrong time because they were responsible to register the time. For subsequent experiments, we intend to use tools that measure the time automatically based on key events. Also, because they were in working time and could have other appointments, some may have automated in a different pace if comparing to a real task.

External Validity: We conducted the analysis with artifacts and personal from a real project. However, products and TCs may vary from project to project, even the way to write TCs, which could affect our algorithm to reuse test actions. To mitigate this, we considered a traditional way to present TCs (description, steps and expected results) that we believe to be flexible enough to apply on most software projects. It is worth mentioning, however, that we only considered tests for the Android platform that could be automated by just screen interactions and were visually checkable. But the strategy proposed in this paper could be easily tested in other platforms (web, mobile) by extending the capture & replay modules, since the reuse strategy is framework-agnostic. Even so, regarding the time to automate, we got similar results to Leotta et. al. (2013) [8] experiment with web frameworks.

VI. Conclusions and Future Work

Test automation based on coding scripts raises some issues: (1) it requires hiring specialized people to automate TCs; (2) code maintenance is not an easy task and it is often required, mainly because it is UI-based; (3) every new TC must be coded by a developer, even when it is similar to

previous ones. To deal with the two former issues, we proposed here a stragety and a tool that captures manual interactions made by testers, and converts them into an intermediate representation (test action) which is framework-independent and can be easily mapped into any automation framework. Only at runtime we interpret the framework script associated with atomic actions. Currently, the tool interprets scripts for *UiAutomator* and for a proprietary framework, and has been applied this to generate tests for verifying mobile devices in an industrial context of a partnership with Motorola Mobility. Because testers themselves can record their interactions, a dedicated automation team is not essential anymore; this eases maintenance because the testers only need to record new interactions on the device. To overcome the third issue, the tool is also capable of translating new TC descriptions into framework scripts by matching each test step with a test action previously saved in a database; as such, the more actions are stored, the better is the reuse ratio, consequently reducing the effort required to automate new TCs.

Unlike other capture & replay tools, AutoMano provides a straightforward way to reuse actions previously recorded by users, searching the database for actions whose associated descriptions are similar to the user input. The evaluation of this strategy brought evidences of a reuse ratio up to 71%, without drawbacks in the automation pace when compared to other capture & replay approaches.

Building upon the results already achieved, our work creates opportunity for several future research directions.

Matching improvements: To improve the matching success rate, we intend to adopt a controlled natural language to describe TCs, instead of using free English. We expect that a controlled grammar will gradually become representative enough to be the test action structure itself. Also, ontology-based representations for automation [23] could be used to express semantic relationships and context.

Capture and Replay improvements: As mobile devices frequently present new input sensors, it is important to cover these sensors by capturing interactions on new ones like voice or gesture motions. Regarding maintenance, we plan to adopt more flexible algorithms that are aware of UI changes, in order to handle minor UI changes.

Exploratory testing: An interesting topic for future investigation is the automatic generation of a large number of new TCs by combining test actions stored in the database, for the purpose of exploratory testing. This can be based both on a random strategy or on a more elaborate guided approach that only combines actions that are compatible. A notion of compatibility can be characterized by extending the structure of test actions with pre- and postconditions so that the sequential composition of two actions is meaningful only if the postcondition of the first action logically implies the precondition of the second.

ACKNOWLEDGMENT

This work is partially supported by CNPq (Grant 132329/2015-8) and Motorola Mobility. We thank Rodrigo Folha and Cesar Albuquerque for help with implementation; Benicio Goulart, Alice Arashiro, Guilherme Almeida, Virgínia Viana and Dacio Mendonça for useful comments and follow-up; and for all testers who gave us feedback.

REFERENCES

[1] I. Burnstein, *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2003.

[2] R. Chandra, B. F. Karlsson, N. Lane, C.-J. M. Liang, S. Nath, J. Padhye, L. Ravindranath, and F. Zhao, "Towards scalable automated mobile app testing," Technical Report MSR-TR-2014-44, Tech. Rep., 2014.

[3] H. K. Leung and L. White, "Insights into regression testing [software testing]," in *Software Maintenance, 1989., Proceedings., Conference on*. IEEE, 1989, pp. 60–69.

[4] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[5] B. Beizer, *Software testing techniques*. Dreamtech Press, 2002.

[6] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 123–133.

[7] C.-T. Lin, C.-D. Chen, C.-S. Tsai, and G. M. Kapfhammer, "History-based test case prioritization with software version awareness," in *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*. IEEE, 2013, pp. 171–172.

[8] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 272–281.

[9] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[10] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[11] W. Grieskamp, "Multi-paradigmatic model-based testing," in *Formal Approaches to Software Testing and Runtime Verification*. Springer, 2006, pp. 1–19.

[12] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 72–81.

[13] CloudMonkey. (2013) Monkeytalk. [Online]. Available: https://www.cloudmonkeymobile.com/monkeytalk

[14] Robotium. (2013) Robotium recorder. [Online]. Available: http://http://robotium.com/

[15] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *Software, IEEE*, vol. 32, no. 5, pp. 53–59, 2015.

[16] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *null*. IEEE, 2003, p. 260.

[17] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "Nat2testscr: Test case generation from natural language requirements based on scr specifications," *Science of Computer Programming*, vol. 95, pp. 275–297, 2014.

[18] M. Esser and P. Struss, "Obtaining models for test generation from natural-language-like functional specifications," *Proceedings of DX*, vol. 7, pp. 75–82, 2007.

[19] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[20] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[21] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[22] V. Caldiera and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, vol. 2, no. 1994, pp. 528–532, 1994.

[23] S. Paydar and M. Kahani, "Ontology-based web application testing," in *Novel Algorithms and Techniques in Telecommunications and Networking*. Springer, 2010, pp. 23–27.