# On Building Test Automation System for Mobile Applications Using GUI Ripping

Chuanqi Tao
Nanjing University of Science and Technology
Nanjing, Jiangsu, China
taochuanqi@njust.edu.cn

Jerry Gao
San Jose State University
San Jose, CA, USA
Taiyuan University of Technology, Taiyuan, China
Corresponding to: jerry.gao@sjsu.edu

**Abstract:** With the rapid advance of mobile computing technology and wireless networking, there is a significant increase of mobile subscriptions. This brings new business requirements and demands in mobile software testing, and causes new issues and challenges in mobile testing and automation. Current existing mobile application testing tools mostly concentrate on GUI, load and performance testing which seldom consider large-scale concurrent automation, coverage analysis, fault tolerance and usage of well-defined models. This paper introduces an implemented system that provides an automation solution across platforms on diverse devices using GUI ripping test scripting technique. Through incorporating open source technologies such as Appium and Selenium Grid, this paper addresses the scalable test automation control with the capability of fault tolerant. Additionally, maximum test coverage can also be obtained by executing parallel test scripts within the model. Finally, the paper reports case studies to indicate the feasibility and effectiveness of the proposed approach.

**Keywords:** mobile application testing; test automation; large-scale concurrent testing; GUI ripping

## I. Introduction

With the rapid advance of mobile computing technology and wireless networking, there is a significant increase of mobile subscriptions. This brings new business requirements and demands in mobile software testing, and causes new issues and challenges in mobile testing and automation. Mobile Application testing refers to testing activities for native and web applications on mobile devices using well-defined software test methods and tools to ensure quality in functions, behaviors, performance, quality of service and features like mobility, usability, inter-operation ability, connectivity, security and privacy [10][22]. Most of the present research work focuses on providing solutions to the technical problems on mobile app white-box testing methods, GUI-based test technique, and ad-hoc scripting test tools;

GUI ripping has become an effective technology in GUI test automation. Test cases and test execution can both be automated using well-defined GUI ripping models [4][14][19][21]. Current existing mobile app tools mostly concentrate on GUI, load and performance testing which hardly considers large-scale concurrent automation, coverage analysis, fault tolerance and usage of well-defined models. The existing test automation approaches and tools suffer from several challenges and problems. The first issue is that it is too costly to deal with diversity of mobile test environments on varieties of mobile devices, and the other is the lack of cost-effective method and platforms to support a unified mobile test automation crossing different mobile platforms on diverse devices. Most of existing tools do not analyze the data dependence between GUI components. In summary, there is a lack of mobile test scripting technique to address large-scale concurrent mobile test needs.

This paper intends to develop a system tool that provides a large-scale automation solution using GUI ripping. By incorporating open source technologies such as Appium and Selenium Grid, we plan to address the scalable test automation control. Additionally, maximum test coverage can also be obtained by executing parallel test scripts within the model. Hence, this approach will be useful in numerous mobile app validation and test automation within the industry. The contributions of this paper can be summarized as follows.

- Test automation solution that would provide services for concurrent test automation for multiple mobile devices running on different platforms.
- Dependency analysis report which helps in identifying the dependency existing between the multiple scripts intended to run on different paths.
- Coverage analysis report after the end of each successful test execution.

The paper is structured as follows. The next section presents the background and related work. The system requirement analysis is introduced in Section III. Section IV presents the designed and implemented prototype tool for the proposed approach. Case studies of testing on sample mobile apps are shown in Section V. Conclusions and future work are summarized in Section VI.

## II. Background and Related Work

### A. Background

The purpose of ripper is to identify maximum number of structural information about the GUI of any android application through the algorithm. Here, the application's windows are opened in depth-first manner. The ripper is capable of extracting all the widgets and its properties from a GUI window. Properties of a window can be size, color, status or window types. GUI windows can be categorized into two types. Model window allows users to fire an event

DOI reference number: 10.18293/SEKE2016-168

from within the window like a Save button. Modeless window on the other hand, let users expand a set of commands like Replace command, which doesn't restrict the user's focus to a specific range of events within a window. Thus, ripper abstracts widgets and their properties and generates a GUI tree.

Hierarchical nature of the GUI is represented in a GUI tree. Each node of the GUI tree is called a window or activity [14], [19]. An Event-Flow Graph (EFG) is generated by taking the input as a GUI tree that is the outcome of applying the ripper algorithm on the application. An EFG is a directed graph epitomizing the GUI events. Each node in the EFG is any GUI event. An edge from node u to node v represents a 'follows' relationship that suggests event v can be executed soon after the event u. Each event in the EFG will contain the widget ID with which, the matching GUI node can be identified through the GUI tree.

### B. Related work

Huang et al. proposed an automated test case generation framework for GUI testing. An accessibility framework known as Microsoft UI Automation (UIA) is used to perform Reverse engineering to generate the event flow graph [1]. Memon et al. discussed GUI test modeling, test generation and replay, and factors for controlling flakiness of generated test cases [2]. The models used for event spacing are state machine and graph models. Extracting, generating and running GUI tests are performed using a tool called GUITAR (GUI Testing frAmewRk). Li et al presented a solution for GUI automation testing for smartphone applications from user behavior models known as AD Automation framework [3]. It supports behavior modeling, automated GUI test case generation, test case simulation and error diagnosis based on UML activity diagrams. Test script library is constructed based on the model and GUI test case generation makes use of these libraries. Also the log analyser performs the error diagnosis after test execution. Amalfitano et al proposed a technique that automates Android mobile applications testing [4]. Amalfitano et al. also presented a tool called AndroidRipper that uses an automated GUI-based technique to test Android apps in a structured manner. It relies on a GUI crawler which crawls through the application GUI to generate test cases. This crawler will then reveal the fault points in the application such as runtime crashes and it can also be used in the regression testing. GUI ripping has become a popular technology in GUI test automation using well-defined GUI ripping models [2, 3]. GUI-based testing has been discussed in numerous papers. For instance, Anand et al. [20] introduced an automated testing approach to validating mobile GUI event sequences for smart-phone apps. Similarly, Amalfitano and his colleagues [21] presented a tool called AndroidRipper that uses an automated GUI-based technique to test Android apps in a structured manner.

Certain research efforts are dedicated to developing tools or frameworks to address limitations in current commercial tools. For example, JPF-ANDROID is a verification tool that supports white-box testing for mobile apps, and JeBUTi/ME is a tool that offers white-box test coverage analysis based on a conventional CFG-based test model. A few recent research papers focus on GUI-based testing using scripts and GUI event-flow models [20][21].

## III. System Requirement Analysis

The main business requirement for the system is test automation which is cost effective and achieves maximum coverage by following the model based testing. We have used open source tools to achieve test automation. GUI Ripper automatically analyses the mobile application by examining the Graphical User Interface (GUI). It uses both standardized exploration strategies such as DFS (Depth First Search) and BFS (Breadth First Search) and random techniques. This results in an abstract GUI model which is also known as GUI tree [19]. An Event Flow Graph can be deduced from this GUI tree graph. Test sequence and test cases are generated based on the Event Flow Graph. This type of model based approach helps to achieve maximum degree of coverage of the application UI.

The description of use case can be illustrated as follows. User accesses the web page and uploads the application which is an *apk* file; A model is generated by the automation server by the GUI ripping process, which the user is able to view in an XML format; The user can also view the dependencies; User gets the tests scripts being generated by the automation server; The test scripts are run by the user with the help of runner component; After the test script execution, user can view the test execution summary; Coverage analysis is provided to the user. Table 1 shows system level functionalities by explaining the basic components available in our system and the corresponding responsibilities. Due to space limitation, other requirements such as non-function, behavior, resource, and etc are not listed here.

**Table 1 Several typical function requirement of the system**

| Components | Responsibilities |
|---|---|
| Emulator/Mobile Environment Setup | Helps during the mobile/Emulator environment setup at deployment |
| GUIRipper | Traverses applications' GUI and generate GUI model |
| EFG | Converts GUI model into event flow graph with XML format |
| TestGenerator | Identifies the action sequence and generating test scripts based on it. |
| CoverageAnalyzer | Responsible for test result analysis and coverage analysis. |

## IV. Design and Implementation for the System

This section presents the detailed software system architecture and explains about the involved components and their relations and connectivity.

### System architecture design

Existing mobile testing tools face several challenges and issues. We intend to design and implement a feasible cost effective solution to model based test automation as depicted in Figure 1. Test automation web application is an interface for users to upload android APK file. It displays model and graph. Users can also view the test execution and test summary using the UI interface through MAC terminal. Test Automation Server takes the uploaded APK file as the input to proceed further with ripping, analyzing dependencies and generating test cases. Test scripts are built based on the grouped scenarios. These test scripts are fed to the test runner module. Test runner passes the test scripts to the selenium grid. With the *appium* nodes running, test scripts are executed on the registered mobile devices and emulators. Model manager stores the path details and generates model in the database and likewise Test Gen and Runner store test scripts and test results respectively.

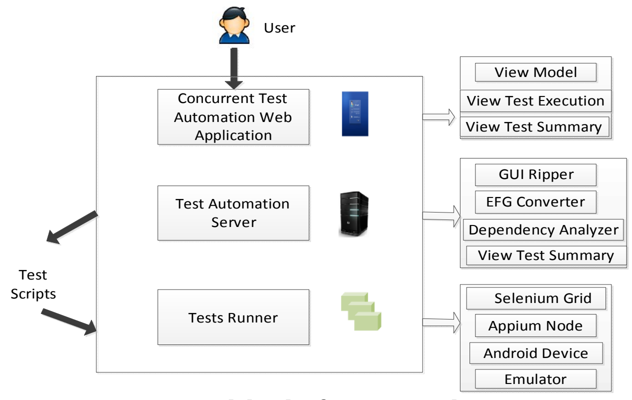


**Figure 1 High level infrastructure diagram**

**Conceptual Framework-** To achieve model based test automation, we designed a test automation server. It is a conceptual framework which includes the interaction between all the components of the automation severs. We have included the sub modules of each component. Parser in the diagram is tool which fetches the elements from the under test application. Using this application data in GUI traverse or GUI ripping algorithms we can generate the test model in the model manager component. We also perform Scenario extraction and dependency analysis using this generated model. Dependencies here are among screens, function etc. Based on the analysis of model manager, a Test Gen component generates the action sequence and test scripts. These test scripts are then fed into Runner component where the actual test execution occurs. Finally the Test Analyzer component does the post-test analysis to generate the coverage report and test summary.

**Centralized Test Automation Platform-** To achieve scalability, we designed a scalable grid node diverse framework. Multiple scripts can run on multiple mobile devices irrespective of the platform and version. Some well-known open source technologies in Selenium grid and Appium are incorporated.

**Test Runner Components-** As depicted in Figure 2, test runner components constitute of selenium grid, appium nodes, android mobile devices and emulators. Selenium grid acts as hub and supports large scale distributed testing. It manages multiple nodes, checks for active and inactive grid and updates the status. Multiple appium nodes can be registered to the hub. It supports mobile application automation testing. Multiple devices and emulators can be incorporated for test automation using selenium hub – appium node configurations.

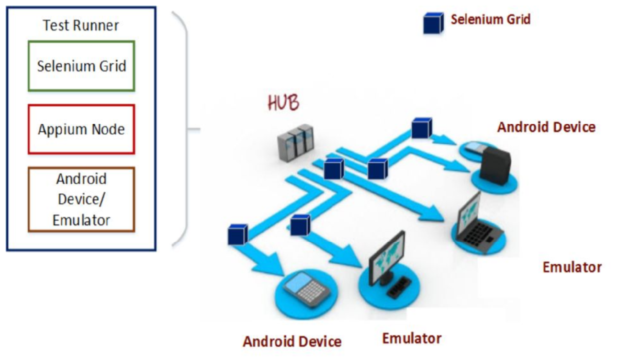


**Figure 2 TestRunner component samples**

**Test Automation Server Components-** We have divided our test automation server into four main modules, i.e., GUI Ripper, Dependancy Analyzer, Test Gen, Runner to serve various purposes like generating test model, analyzing dependencies, generating testing cases and executing test scripts.

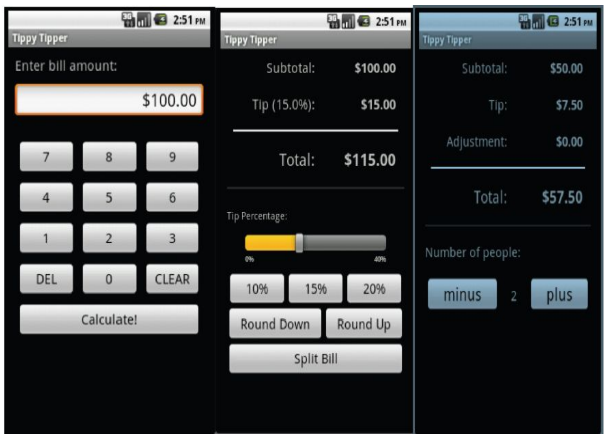


**Figure 3 TippyTipper sample android application**

## V. Case Studies

### A. Study object

To apply the developed tool for mobile application GUI testing, we used several realistic mobile applications and systems. Here we choose TippyTipper as the sample android application to illustrate our approach. This is a simple application to calculate the tips for a meal. Figure 3 illustrates the simple functionalities existing in the application. This application has 3 windows. In the first window, it allows the users to enter the bill amount with numeric keypad. The DEL button deletes a rightmost (to the cursor) numeric number and CLEAR button clears the number entered in the edit text. The CALCULATE button allows the user to move into the second window. Second window displays total bill amount including the calculated tax amount. It also has the progress bar with which you can modify the percentage of tax you want to calculate (0-100), option to roundup and split the bills. On pressing the SPLIT BILLS, the control goes to the third window, which has the option to modify the number of users within the bill has to be shared.

Below subsection shows the detailed approach we followed for automated testing of TippyTipper application.

## B. Model generation

We have used the open source GUI ripper algorithm for ripping the GUI. This will result in a file with its GUI components which is basically a GUI tree. This is later fed into the converter, which eventually produces our model as Event Flow Graph.

The purpose of Ripper is to identify maximum number of structural information about the GUI of any android application through the algorithm. Here, the application's windows are opened in depth-first manner. The ripper is capable of extracting all the widgets and its properties from a GUI window. Properties of a window can be size, color, status or window types. GUI windows can be categorized into two types. Model window allows the user to fire an event within the window, like a Save button. Modeless window on the other hand, allows the user to expand a set of commands like Replace command, which doesn't restrict the user's focus to a specific range of events within a window. Thus, ripper abstracts widgets and their properties and generates a GUI tree. Each EFG will have an adjacency matrix which helps in identifying the edge between two nodes. If the value is 0, there is no edge between 2 nodes. If the value is 1, there exists an edge. If the value is 2, there exists an edge to itself.

In our implementation, the pictorial representation of the graph is drawn with the help of the adjacency matrix present in the EFG file. The *GraphViz* tool is used for displaying the graph. The EFG file is first converted into dot file format, and then fed into the *GraphViz* which generates the nodes and edges based on the adjacency matrix.

## C. Dependency analysis

Dependency analysis is required in order to achieve the concurrency of test automation. The dependent nodes are clustered in order to identify the test cases to be executed in different cycles. To generate a cluster, loops within the graph have to be identified. Hence, the input to this component would be the EFG generated through ripper.

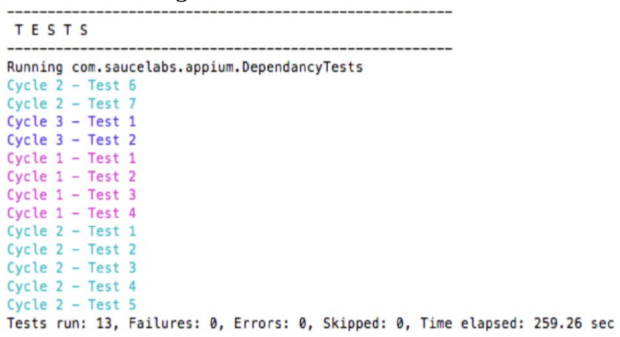
The dependency can be identified w. r. t features, data or function. We are planning to implement a high level dependency based on the connectivity existing within the application based on the EFG. However, this dependency does not provide the complete dependency existing within the model. Therefore, we are using the Donald B. Johnson algorithm to identify the elementary cycles in a graph at first. This is based on the search for the strongly connected components within the graph. The two important modules of the program are given below. This piece of code identifies the list of objects that contains list of nodes of all elementary cycles in the graph.

## D. Test case generation

The application components are nested and hidden once the application is launched. This makes the task of test case generation tedious and time consuming. Thus, the Event flow graph is used to generate the test case for AUT. The test criteria for generating the test cases have been kept that each event in the EFG covers at least 5 test cases. The graph traversal method is used to traverse through the EFG to identify the starting main screen and to track the count of the traversal. The maximum limit for the events in a single event sequence has been kept as 50. The test case inserts the connecting events in the test case to make it executable on the real GUI.

## E. Test script runner

Test script runner takes the generated test cases and the model as the input. For each test case, the runner checks the corresponding edges and events in the GUI tree and EFG model for knowing the window and widget information on which the events are executed. The Test runner constitutes of Selenium Grid, Appium nodes, Android Devices and emulators. To begin with the runner process, selenium is first registered as a hub to control all the nodes configured. After successful launch of selenium grid server, appium nodes are registered at different ports. Each appium node has its own configuration.



```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.saucelabs.appium.DependancyTests
Cycle 2 - Test 6
Cycle 2 - Test 7
Cycle 3 - Test 1
Cycle 3 - Test 2
Cycle 1 - Test 1
Cycle 1 - Test 2
Cycle 1 - Test 3
Cycle 1 - Test 4
Cycle 2 - Test 1
Cycle 2 - Test 2
Cycle 2 - Test 3
Cycle 2 - Test 4
Cycle 2 - Test 5
Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 259.26 sec
```

**Figure 4 Test results samples**

The node configuration includes capabilities of the appium node. The browser name, version, platform etc can be defined here. The configuration part includes the hubHost, which is the ip address of the machine on which selenium grid is running. The ip address of the hub Host and URL are where the machine appium is running on. The hub Port is 4444 by default, where the selenium grid is listening for requests. Different appium nodes can be registered at different ports by assigning different port numbers.

Based on the data dependancies, test scripts are grouped into cycles. Each cycle has test scripts based on the paths for the dependant node. As shown in Figure 4, test results are shown after execution of all test cycles. For example, Cycle 2 covers test scripts for one node whose date is dependent on other nodes. Hence, different cycles are color coded.

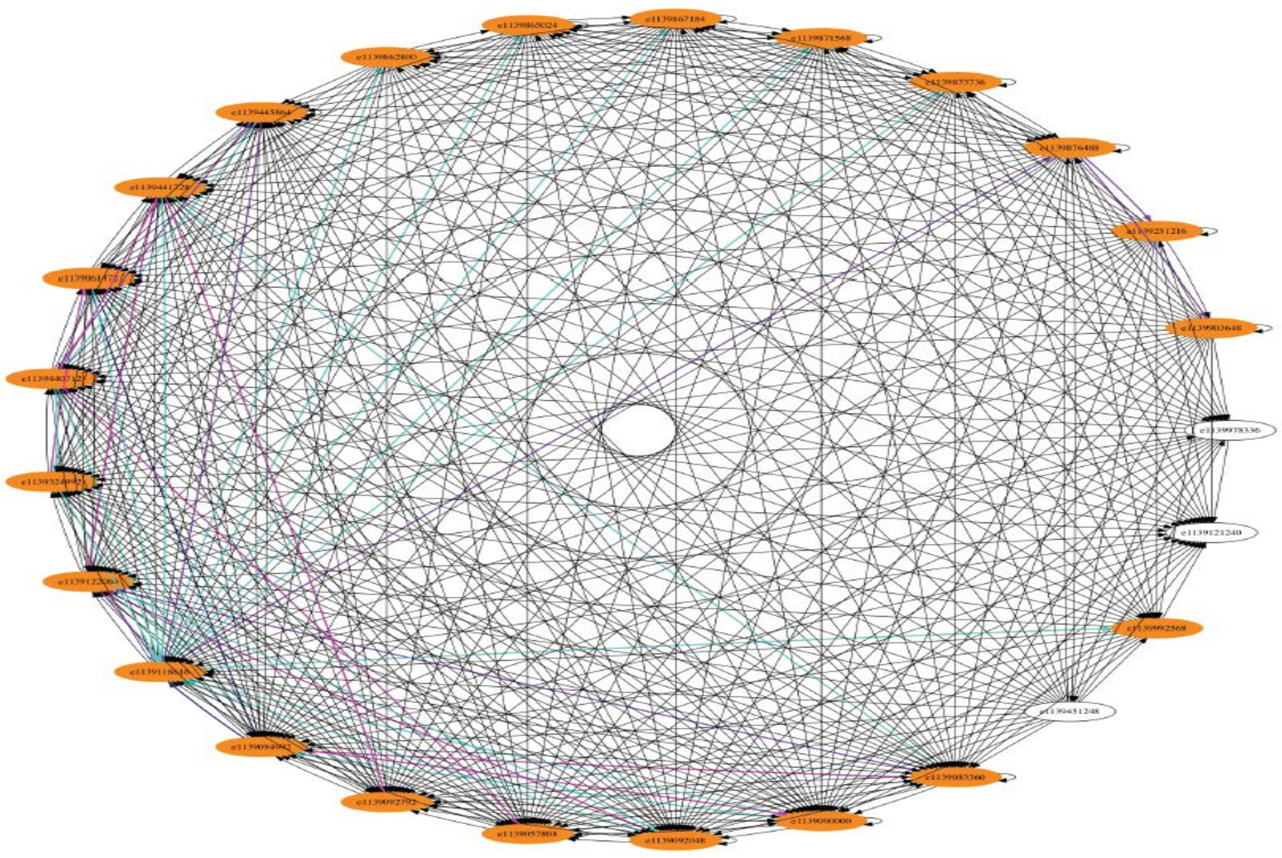


**Figure 5   Event Flow Graph with node coverage analysis**

As depicted in Figure 5, nodes are reached by finding their xpath. The Figure shows the tests results after execution of all test cycles. Each test-script determines a dependant path. All possible node paths covered are color coded in the graph based on the cycles. For cycle 1 the color code is Magenta, cycle 2 it is Cyan and cycle 3 is Purple. For any test script that fails the color code is red. The tippy tipper app has 25 nodes altogether. We achieved 92% node coverage by covering 22 nodes through our test scripts as depicted in Figure 5. The nodes covered are colored in orange. The uncovered paths are in depicted in black.The color codes are illustrated as follows.

*Magenta* - *Cycle 1 Test scripts – Scenarios covered*
*Cyan – Cycle 2 Test scripts – Scenarios covered*
*Purple – Cycle 3 Test scripts – Scenarios covered*
*Black – Scenarios not covered*
*Orange – Covered nodes*

## F.  Limitations, and Experience Learned

The proposed approach suffers from some limitations. For example, the syntax is currently dependent on variables. Thus if the variables associated change, the step definitions need to be modified. As a complex problem in the known space, the new id needs to be extracted from the app properties. Therefore, there is tremendous scope to automate and reduce this dependency on the variables in the future work.

Initially, to set up ith the infrastructure, we planed to support test automation where in the hub manager should be robust enough to back and forth data between nodes and test servers/database. Also, since diverse platforms are supported, the same test script written should work on mobile devices running on multiple platforms. Additionally, test scripts are run on multiple devices simultaneously so that scalability is achieved. Furthermore, it has to be taken care that the final solution is fault tolerant, i.e., if a single script fails in one of the devices, it should not fault the other scripts running on the device. Nevertheless, it should not stop the scripts running on other mobile devices. Hence, the automation solution has the challenge of building concurrent, scalable, fault tolerant, unified mobile test automation tool.

## VI.  Conclusions and Future Work

This paper specified the requirements, design, schedule and budget plan related to our project from the industry perspective. All the team members contributed equally in the entire process and we had great scope for learning from each other. In this paper we aim to develop a tool that performs concurrent test automation for mobile applications. We will make use of open source tools to achieve our aim through this project. The approach aimed at solving several challenges faced by the existing tools in the market. The achieved solution provides GUI test automation of android apps based on model and data dependencies.

Currently our approach supports a unified central test automation platform for only Android OS. It can be further enhanced by adding GUI ripper for iOS apps. This way it can be extended to iOS OS as well. In addition to this, the test runner can be deployed on Amazon EC2 or OpenStack to provide efficient approach for scalability. By then scalability can be achieved at grid level, which is at node level currently.

## References

[1] Y. Huang and L. Lu, "Apply ant colony to event-flow model for graphical user interface test case generation," IET.Software, pp.50-60, Feb.2012.

[2] A.M. Memon and M.B. Cohen, "Automated testing of GUI applications: Models, tools, and controlling flakiness," in 2013 Int.Conf. Software Engineering (ICSE), pp.1479-1480.

[3] A. Li, Z. Qin, M. Chen and J. Liu, "ADAutomation: An Activity Diagram Based Automated GUI Testing Framework for Smartphone Applications," in 2014 Int. Conf. Software Security and Reliability, pp. 68-77.

[4] D. Amalfitano, A.R. Fasolino and P. Tramontana, "A GUI Crawling-based technique for Android Mobile Application Testing," in 2011 Int. Conf. Software Testing, Verification and Validation Workshop (ICSTW), pp. 252-261.

[5] V. Gaur, V. Ragunathan and V. Prakash (2011), "Mobile Test Automation Solutions," Hexaware Technologies., Jamesburg, NJ, Mobile Accelerator White Paper, Available on http://hexaware.com/casestudies/hs-it-wp-1.pdf.

[6] Testing Strategies and Tactics for Mobile Applications, Keynote Systems, Inc., White Paper, Available at http://www.keynote.com/resources/whitepapers/testing-strategies-tactics-for-mobile-applications.

[7] A. Memom, I. Banerjee and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in 2003 Working Conf. Reverse Engineering (WCRE), pp. 260-269.

[8] J. Bo, L.Xiang and G. Xiaopeng, "Mobile Test: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices," in 2007 Int. Workshop Automation of Software Test, pp. 8.

[9] P. K. Govindasamy(2012), "Selecting the Right Mobile Test Automation Strategy: Challenges and Principles,", cognizant 20-20 insights Inc., Available at http://www.cognizant.com/InsightsWhitepapers/Selecting-the-Right-Mobile-Test-Automation-Strategy-Challenges-and-Principles.pdf.

[10] J. Gao, X. Bai, W. T. Tsai, T. Uehara, "Mobile App Testing – A Tutorial", IEEE Computer – Special Issue on Software Validation,Vol 47, No.2, pp.46-55, 2014.

[11] J. Gao, C.C.Y. Toyoshima, D.K. TLeung, "Engineering on the Internet for Global Software Production", Computer journal, Vol 32, No.5, pp. 38-47, May 1999.

[12] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, J. Kazmeier, "Automation of GUI testing using a model-driven approach", International workshop on Automation of software test, pp.9-14, 2006.

[13] J. Gao, C. Tao, "Modeling mobile application test platform and environment: testing criteria and complexity analysis", Industry Contributions to Test Automation and Model-Based Testing, pp.28-33, 2014.

[14] A. M. Memon, M. E. Pollack, M. L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning", IEEE Transactions on Software Engineering - Special issue on 1999 international conference on software engineering, Vol 27, No.2, February 2001.

[15] C. Siemens, "The Search for Mobile App Test Automation", blog, November 19, 2013, Available at http://engineering.zillow.com/the-search-for-mobile-app-testautomation/

[16] Test Automation Tools for Mobile Applications: A brief survey, white paper, HSC PROPRIETARY, Available at http://www.hsc.com/Portals/0//Uploads/Articles/hsc_whitepaper_mobileTestAuto mation_22Feb2013634971268468845610.pdf

[17] Sebastian Bauersfeld, A Metaheuristic Approach to Automatic Test Case Generation for GUI-Based Applications, 22 August 2011, available at http://www2.informatik.huberlin.de/swt/dipl/bauersfeld-2011.pdf

[18] G.J. Cong, D. Bader, Lockfree Parallel Agorithms: An experimental Study, available at, http://www.cc.gatech.edu/~bader/papers/lockfree-HiPC2004.pdf.

[19] A. Memom, I. Banerjee, B. N. Nguyen, B. Robbins. The first decade of GUI ripping: Extensions, applications, and broader impacts, in Working Conf. Reverse Engineering (WCRE 2013), pp 11-20.

[20] S. Anand et al., "Automated Concolic Testing of Smartphone Apps,"Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng. (FSE12), 2012, pp. 1–11.

[21] D. Amalfitano et al., "Using GUI Ripping for Automated Testing ofAndroid Applications," Proc. 27th IEEE/ACM Int'l Conf. Automated SoftwareEng. (ASE 12), 2012, pp. 258–261.

[22] H. Muccini, A. D. Francesco, and P. Esposito, "Software Testing of Mobile Applications: Challenges and Future Research Directions", Proceedings of International Workshop on Automatic Software Test Automation, 2012, pp 29-35.

## Acknowledgement

This paper is supported by the National Natural Science Foundation of China under Grant No.61402229 and No.61502233; the Open Fund of the State Key Laboratory for Novel Software Technology (KFKT2015B10), and the Postdoctoral Fund of Jiangsu Province under Grant No.1401043B.