CROSS-PLATFORM TESTING AND MAINTENANCE OF WEB AND MOBILE APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Shauvik Roy Choudhary

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
May 2015

CROSS-PLATFORM TESTING AND MAINTENANCE OF WEB AND MOBILE APPLICATIONS

Approved by:

Dr. Alessandro Orso, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Mayur Naik
School of Computer Science
Georgia Institute of Technology

Dr. Russell J. Clark
School of Computer Science
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Mukul R. Prasad
Research Manager
Fujitsu Laboratories of America

Date Approved: 4 December 2015

To my parents,

Swagata and Rajkumar,

for their continuous love and support.

ACKNOWLEDGEMENTS

I am thankful to the following people for their guidance, support and company throughout my graduate school.

First and foremost, I would like to thank my advisor, Dr. Alessandro Orso, for 8 years of his guidance, patience and encouragement; first during my masters, followed by my PhD years at Georgia Tech. I still remember the day (August 16th 2007) when I first knocked on his door to ask if I could work with him. Alex's consent changed my career and I will always be indebted to his generosity. He challenged me to excel both in research and in presenting my work to the community. At every stage of my PhD, Alex has inspired me by setting an example through hard work, and has supported my various endeavors, including my attempts at multiple startup programs.

I express deep gratitude to Dr. Mukul Prasad for his kindness, guidance and support during my internship and for the collaboration that flourished afterward. Mukul's mentorship has been invaluable for me. He has always been supportive of my work during my PhD and has given me feedback to steer it towards the right direction. Under Mukul's guidance, I gained a broader view and liking for industry research and realized the importance of making research tools useful for real developers.

I would like to thank other members of my committee, Dr. Mayur Naik, Dr. Russell Clark and Dr. Milos Prvulovic, who have helped me make this dissertation better, through their counsel and support. I am also thankful to other faculty members in the SE group, especially *late* Dr. Mary Jean Harrold and Dr. Spencer Rugaber, who gave me helpful suggestions during the initial days of my PhD.

My PhD would not be as joyful without my labmates. I would like to thank Dr. Sangmin Park and Wei Jin for their dear friendship and company throughout the

years, Mattia Fazzini for getting me excited about espresso (both the drink and the testing framework), and other members of my lab for their camaraderie.

The former students from the group have been immensely helpful at times when I needed advice. Dr. GJ Halfond was kind enough to advise me through a collaboration for my masters project. It was while working with GJ, that I developed my interest for research, and decided to apply to the PhD program. Dr. James (Jim) Clause taught me the secret of structuring a paper and creating elegant presentations. Other senior students in the group, especially Dr. Saswat Anand, Dr. Chris Parnin and Dr. Hina Shah, have shared their valuable insights from their research experiences.

My friends have played an essential role in keeping my life at decent levels of sanity. Tanushree Mitra has unconditionally accompanied me through the most challenging years of my PhD. I cannot thank her enough for her patience and thoughtfulness. Pushkar Kolhe, my roommate and a close friend has been by my side through the highs and lows of my graduate school days. My friends, Tushar Kumar and Nawaf Almoosa have had a very positive influence on my social life with their delightful company. I would also like to thank Partha and Tanushree Chakraborty, who have been equivalent to elder siblings. I am grateful to Shafi and Adria Motiwalla, who have been like family away from home for many international students, like me. I convey a big thanks to all friends, who have left an everlasting impression on me.

Last but not the least, I would like to thank my family. My little sister, Shatabdi, who continues to impress me with her enthusiasm and smartness. My parents, Swagata and Rajkumar, who have encouraged me to pursue graduate school and have been unconditionally supportive of all my endeavors. My mentor and uncle, Dr. Sugata Sanyal, has appreciated my passion towards computer programming since my high school years. I am fortunate to be born in a little village in West Bengal, India, where most of my extended family still resides. A final thanks to all of them, especially my grandparents, for their values, love and wholehearted support through these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Modern software applications need to run on a variety of web and mobile platforms with diverse software and hardware-level features. Thus, developers of such software need to duplicate the testing and maintenance effort on a wide range of platforms. Often developers are not able to cope with this increasing demand and release software that is broken on certain platforms, thereby affecting a class of customers using such platforms. Hence, there is a need for automating such duplicate activities to assist the developer in coping with the ever increasing demand. The goal of my work is to improve the testing and maintenance of cross-platform web and mobile applications by developing automated techniques for comparing and matching the behavior of such applications across different platforms.

To achieve this goal, I have identified three problems that are relevant in the context of cross-platform testing and maintenance: 1) automated identification of inconsistencies in the same application's behavior across multiple platforms, 2) detecting features that are present in the application on one platform, but missing on another platform version of the same application, and, 3) automated migration of test suites and possibly other software artifacts across platforms. I present three different scenarios for the development of cross-platform web and mobile applications, and formulate each of the three problems in the scenario where it is most relevant. To address and mitigate these problems in their corresponding scenarios, I present the principled design, development and evaluation of the two techniques, and a third preliminary technique to highlight the research challenges of test migration. The first technique, X-PERT identifies inconsistencies in a web application running on multiple

web browsers. The second technique, FMAP matches features between the desktop and mobile versions of a web application and reports any features found missing on either of the platform versions. The final technique, MigraTest attempts to automatically migrate test cases from a mobile application on one platform to its counterpart on another platform.

To evaluate these techniques, I implemented them as prototype tools and ran these tools on real-world subject applications. The empirical evaluation of X-pert shows that it is accurate and effective in detecting real-world inconsistencies in web applications. In the case of FMAP, the results of my evaluation show that it was able to correctly identify missing features between desktop and mobile versions of the web applications considered, as confirmed by my analysis of user reports and software fixes for these applications. The third technique, MigraTest was able to efficiently migrate test cases between two mobile platform versions of the subject applications.

CHAPTER I

INTRODUCTION

The proliferation of cloud and mobile computing has given rise to a diverse set of computing platforms [5, 59]. Consumers use these different platforms for both personal and business activities such as communication, banking, and shopping. To reach these consumers, modern software applications need to run on a wide range of platforms, mainly web and mobile, and present similar functionality on these platforms. However, this implies that software developers need to duplicate their effort for developing, testing, and maintaining their applications on multiple platforms. Although, there are several development approaches that companies use to target multiple platforms, all of them result in software that requires substantial manual effort for testing and maintenance across the supported platforms. Moreover, due to the increased reliance on manual work, such software is often released with cross-platform issues, which results in software failure on affected platforms. This not only causes inconvenience to the users on the affected platforms, but also leads to increased customer support costs and lost revenue for the companies who own such software. Thus, it is essential to study these problems in the most relevant context, and to improve the state of the art using automated techniques.

1.1 Cross-Platform Testing and Maintenance Problems

1.1.1 Identification of Cross-platform Inconsistencies

An important problem in this domain is to identify inconsistencies arising due to the difference in the application's behavior when it is run on two different platforms. In the case of web applications, these inconsistencies can be observed when the web

(a) Firefox (b) Internet Explorer

Figure 1: Issue on Georgia Tech's website on two web browsers.

application is accessed on different web browsers. This results in *Cross-Browser Incompatibilities* (XBIs), which are discrepancies between a web application's appearance, functionality, or both, when the application is run on two different web browser environments. An instance of such a problem is presented in Figure 1, which shows a section from the front page of my institutional website, `http://www.gatech.edu`, on two web browsers, *i.e.,* Firefox and Internet Explorer. As shown in the figure, the layout of the elements on the web page is affected due to differences between the two browsers. In practice, XBIs range from such layout defects to critical problems in the functionality of the web application, and affects all users with a particular web browsing platform.

Due to the increasing popularity of web applications, and the number of browsers and platforms on which such applications can be executed, XBIs have become a serious concern for organizations that develop web-based software. For example, a search on the popular developer discussion forum `stackoverflow.com`, for posts tagged with "cross-browser" returned over 2500 posts over the past four years! Further, nearly 2000 of these have been active over the past year [60].

Because of the relevance of XBIs, a number of tools and techniques have been proposed to address them. In fact there are over 30 tools and services for cross-browser testing currently in the market [13, 45, 55]. Most of these tools are mainly manual and either provide tips and tricks for developers on how to avoid XBIs or render the same web application in multiple browsers at the same time and allow a human to check such renditions. Being human intensive, these techniques are less than ideal; they are costly and, especially, error-prone.

Researchers have therefore started to propose automated techniques for XBI detection (*e.g.,* [11, 19, 45, 51, 53]). At a high level, these automated techniques work as follows. *First*, they render (and possibly crawl) the given web application in two different web browsers of interest and extract a possibly large set of attributes that characterize the application. This set may include behavioral attributes, such as finite state machine models that represent how the web application responds to various stimuli (*e.g.,* clicks, menu selections, text inputs). The set of attributes may also include visual characteristics of certain widgets or sets of widgets on a page, such as their size, their position, or properties of their visual rendition (*i.e.,* appearance). *Second*, the techniques compare the attributes collected across the two browsers and, if they differ, decide whether the difference is attributable to an XBI. Intuitively, these attributes are used as proxies for the human user's perception of the page and its behavior. Thus, differences in attributes between two browsers are indications of possible XBIs. *Finally*, the techniques produce reports for the web-application developers, who can use the reports to understand the XBIs, identify their causes, and eliminate such causes.

The two most fundamental characteristics of XBI detection techniques are therefore (1) the choice of which attributes to collect and (2) the criteria used to decide whether a difference between two attributes is indeed the symptom of an XBI (*i.e.,* it can be perceived by a user as a difference in the web application's behavior or

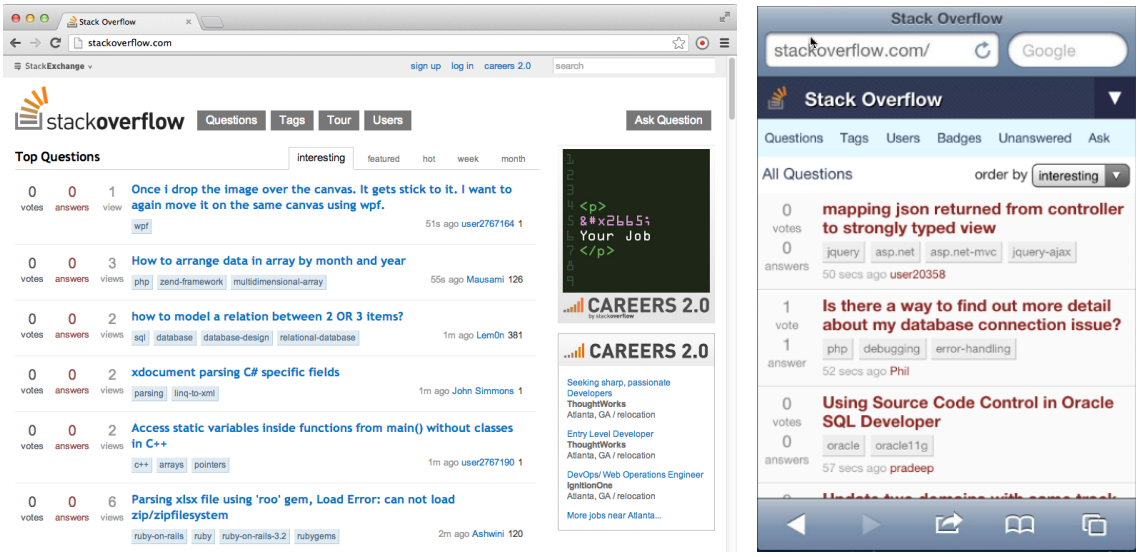
appearance). In existing techniques, these choices are based primarily on intuition and experience and not on a systematic analysis of real-world XBIs.

Although such an approach is fine for an initial investigation, and in fact provided encouraging results in the initial evaluations (*e.g.,* [45, 53]), it must be improved for a more mature solution to the XBI detection problem. Case in point, the evaluation of earlier approaches on a more extensive set of web applications generated a considerable number of false positives, false negatives, and duplicate reports for the same underlying errors. Hence, a principled technique is needed to better detect such XBIs by identifying the most relevant XBI symptom for each kind of XBI.

1.1.2 Detecting missing features between two versions of a multi-platform application

Another common problem in a cross-platform setting is to find missing features between versions of an application, which are developed to target different platforms. Since, parts of a cross-platform application might be significantly different, and often developed separately by different teams, it is common to have features, which are missing on one of the platforms. This problem is indeed relevant across the desktop and mobile versions of a web application. Due to the proliferation of mobile computing devices, it is common practice for companies to build mobile-specific versions of their existing web applications to provide mobile users with a better experience. This customization is necessary, despite the inherently multi-platform nature of web applications, due to the unique features of mobile devices, such as their form factor, user interface, and user-interaction model [66]. Developers thus commonly re-target their web applications, sometimes substantially, to make them more suitable for mobile platforms [26].

In spite of the inherent differences between desktop and mobile platforms, and the resulting differences between desktop and mobile versions of a web application,



(a) Desktop (b) Mobile

Figure 2: StackOverflow.com on desktop and mobile. Although functionally similar, screen-level differences are intentionally added by the developer on mobile.

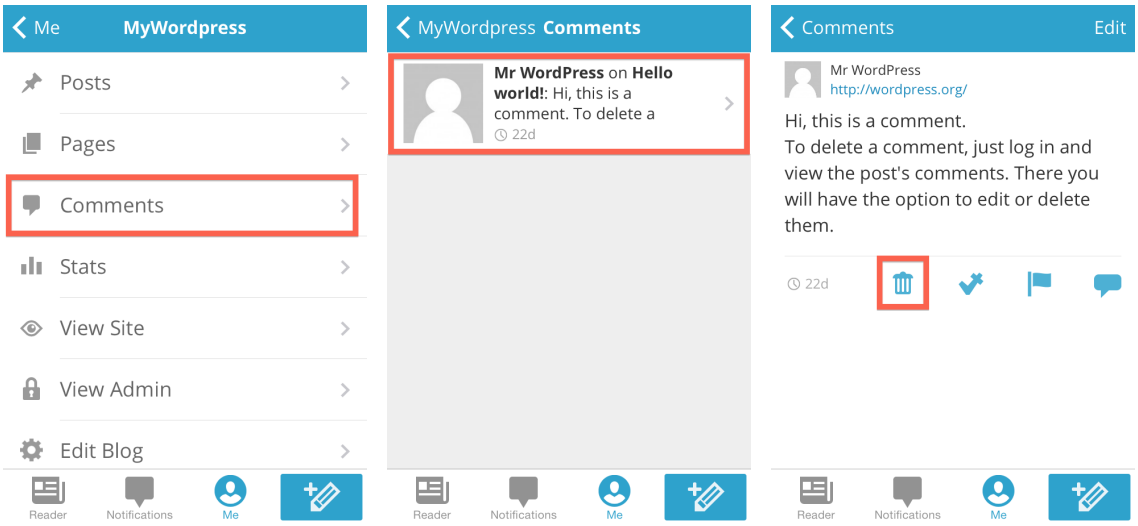
the end user expects some level of consistency in the feature set offered by the application across all platforms. The World Wide Web Consortium (W3C) standards committee, for instance, recommends the "One Web" principle for web browsing platforms [67], which stipulates that web application users should be provided with the same information and services irrespective of the device on which they are operating. Prominent web service providers such as Google [28] and Twitter [64] now follow this guideline, and Figure 2 provides an illustrative example involving the desktop and mobile versions of the popular developer discussion forum `stackoverflow.com`. Although there are substantial differences in the look and feel of the website in the two versions, both versions share the same core functionality: clicking on a question shows detailed information for that particular question in both versions, both versions allow the user to sort the questions according to different criteria (using tabs in one case and the *order by* drop-down menu in the other), and so on.

In this context, the challenge for web developers is to develop different versions of their applications, which are customized to suit the specific characteristics of the

different platforms, and yet provide a consistent set of features and services across all versions. To accomplish this, one common strategy used by developers is to create separate front-end components for desktop and mobile platforms, while keeping (as much as possible) the same server-side implementation [26].

Despite the existence of several libraries and frameworks for helping with this task (*e.g.,* jQuery Mobile [35], Twitter Bootstrap [65], or Sencha [58]), and even tools for migrating existing web application to mobile-friendly versions (*e.g.,* Mobify [48] or Dudamobile [23]), developers perform much of these customizations by hand, which is time consuming and error prone. Furthermore, the different customized versions must also be evolved in parallel, during maintenance, which creates additional opportunities for introducing inconsistencies. As a result, it is often the case that different versions of a multi-platform web application provide different sets of features. Some of these differences are introduced on purpose because of the nature of the different platforms. Location-based features, for instance, are normally available on the mobile version of a web site but not on its desktop version. Some other differences, however, are unintentional and can negatively affect the user experience. This problem is confirmed by the numerous user reports and complaints that appear on the forums for many popular web sites. To illustrate with a concrete example, some users of the popular Wordpress web site (`http://wordpress.org/`) were so frustrated with the problem of missing features on the mobile version of the site (*e.g.,* the inability to upload media files) that they were ready to stop using the software altogether (see Section 4.5).

Hence, it is essential to help developers check their applications to understand how features are implemented across the different platforms. This will not only provide traceability across the platforms but mainly help them track feature completeness.



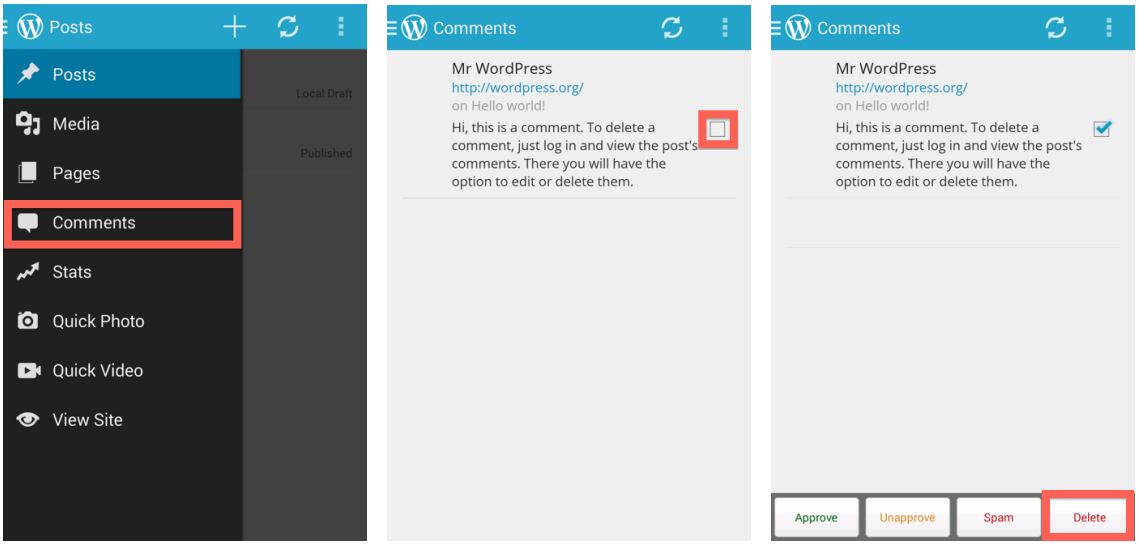
(a) Click menu item (b) Click on comment (c) Click on delete button

Figure 3: iOS: Wordpress Test Script for Deleting a Comment.

1.1.3 Application test migration between two platforms

Another problem arising in a cross-platform setting is to migrate test cases and possibly other software artifacts from one platform to another. This problem is most relevant when the two platforms are significantly different and the application versions for each of these platforms are built separately. An example of this scenario manifests in the case of applications built for the Android and iOS mobile platforms. As confirmed by several developers [3], these applications are developed by different teams and nearly all testing and maintenance tasks are repeated for the different applications. Helping the developers automate some of these tasks will make them more efficient. For instance, automatically migrating tests that are written for the iOS version of the application to the Android version or vice versa, would help them finish the migration task efficiently. Moreover, developers can spend their time on alternate tasks, which require creative human intelligence instead of repeating the test authoring task for each new platform.

However, migrating tests across mobile platforms is challenging for two reasons. Firstly, the application version for each platform is inherently different since they are



(a) Click menu item (b) Select comment (c) Click on delete button

Figure 4: Android: Wordpress Test Script for Deleting a Comment.

developed using different technologies and potentially by different teams. Thus, any technique directed for test migration needs to operate in the face of these differences to find high level similarities. As an example, Figures 3 and 4 show a test script for both the iOS and the Android versions of the Wordpress mobile application, which is used to administer a remote blog. The test case shown in the example deletes a comment on the blog on each platform. The script involves three steps: 1) navigating to the "Comments" page, 2) selecting the comment, and 3) selecting the delete action to complete the test script. On each platform, each of these steps are performed by different set of widgets, which embed different design elements in the page structure. Hence, it is non-trivial for any automated technique to find corresponding actions on widgets across the two platforms. Secondly, the state space of the application on each platform could be extremely large with multiple sets of actions, which could be performed from the current screen. Thus, any matching technique needs to work within these constraints and should aim to maximize the number of test cases migrated from one platform to the other.

1.1.4 Other problems

Other software testing and maintenance problems can also arise in this cross-platform context. Test generation and test selection to target an application migrated to a new platform are interesting research problems, which are currently addressed manually by developers. Another set of challenging problems is to test for non-functional characteristics, such as security and performance, exhibited by the application across platforms. Any issues in these characteristics can be problematic for the user of the application and hence, the developer is expected to find and fix such issues before the application is released. Although these problems are interesting, a technique addressing them needs to be designed to operate with the current developer work flow and replace these manually performed operations with automation. To aid with this design, future researchers can leverage the differential scenario between multiple platforms to address these problems in a way similar to the work presented in this thesis.

1.2 Thesis

A key insight underlying my research is that establishing similarities and differences in application behavior across multiple platforms can be leveraged to address cross-platform problems. However, this behavior can be significantly different and establishing an exact equivalence is infeasible in general. Hence, matching techniques should leverage approximation algorithms to overcome these differences, and thereby address the cross-platform issues.

The thesis of my work is that such approximate behavior matching techniques can be used to automate the testing and maintenance of cross-platform applications, by: 1) uncovering inconsistencies in the behavior of a web application when executed on different browsing platforms, 2) finding missing features between different multi-platform versions of a web application, and, 3) translating test suites, and possibly

other existing software artifacts from one platform version of a mobile application to another.

1.3 Overview of Approach

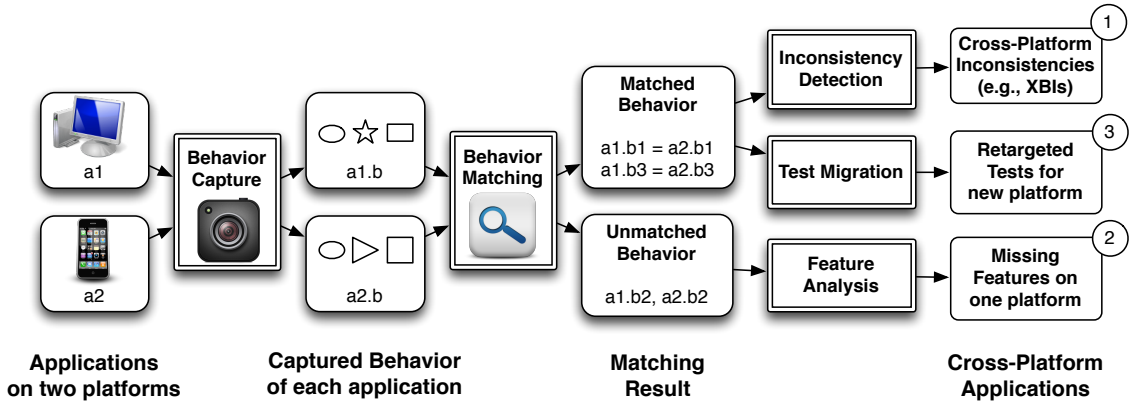


Figure 5: Overall research overview.

To achieve the goal of my thesis, in my research, I present different techniques to capture and match the behavior of cross-platform applications. Figure 5 shows the overall view of my research. As shown in the figure, the approach first captures the behavior of the application running on two different platforms ($a1$ and $a2$). This behavior is represented in the figure by different shapes and is labeled as $a1.b$ and $a2.b$ for the two platforms respectively. The details of this dynamic behavior information vary based on the problem at hand. It can be an explicitly stated model or can be the runtime trace of the application.

Once suitable behavior information is captured, it is then compared across multiple platforms to establish a correspondence between the sets of such information across the platforms. This comparison can lead to both matched and unmatched behavior across the platforms. In the abstract example shown in the figure, consider a matching function, which matches the number of edges in the shapes. Hence, the ovals with infinite edges, ($a1.b1$, $a2.b1$), and the quadrilaterals with four edges ($a1.b3$, $a2.b3$), are assigned to the matched behavior. The star with ten edges, ($a1.b2$), and

the triangle with three edges ($a2.b2$), are assigned to the unmatched behavior set. The behavior matching function shown in this example is rather simplistic. However, in practice the behavior is significantly different and a custom approximate matching technique needs to be developed for each type of behavior to reveal interesting properties relevant to the problem at hand.

Solutions to each of the problems discussed in the previous section can use the behavior matching result in a different fashion to accomplish its goals. For identifying potential cross-platform inconsistencies, matched behavior can be inspected in a finer level of granularity. To address this issue for web applications, I developed the X-PERT technique (Cross-Platform Error ReporTer), which identifies and reports XBIs (c.f. Section 1.1.1). Another application of such matched information is to use the correspondence between the platforms to aid maintenance tasks, such as migration of test cases and possibly other artifacts across the two platforms. A concrete solution to address this problem for native mobile applications is presented in this thesis in the development of the MIGRATEST technique. (cf. Section 1.1.3). Finally, the set of unmatched behaviors indicate the specific instances of application behavior, which are missing on either of the platforms. Analyzing these further can be used to assist the developer in assessing feature completeness while she is developing the application on one of the two platforms. I present a solution for this problem in the context of desktop and mobile web applications in Section 1.1.2 of this thesis, as a part of the FMAP technique (Feature Matching Across Platforms).

1.4 Contributions

My research is developed along the lines of the overview as presented in Section 1.3 and provides the following novel contributions:

- X-PERT — A technique which automatically identifies Cross-Browser Incompatibilities (XBI) in web applications.

- FMAP — A technique to automatically find missing features across the desktop and mobile versions of a web application.
- Empirical evaluation of X-PERT and FMAP on real-world applications to demonstrate their effectiveness.
- MIGRATEST — A preliminary technique to migrate test cases between versions of a native mobile application on two platforms, along with a list of research challenges for future work.

1.5 Organization

The proposal is organized as follows: Chapter 2 presents background details of the three different development scenarios for modern multi-platform applications, and how the problems manifest in these scenarios. Chapters 3 and 4 describe details of the X-PERT and FMAP techniques, along with their empirical evaluation. Chapter 5 describes the MIGRATEST and the research challenges for mobile test migration. Chapter 6 presents related work and Chapter 7 concludes the proposal with a summary of the research along with its contributions.

CHAPTER II

BACKGROUND

In the following sections, we will first define the essential components of a platform. Then we will outline the different development approaches taken by companies to target multiple platforms and will state the inherent testing and maintenance issues faced by employing them.

2.1 Multiple platforms

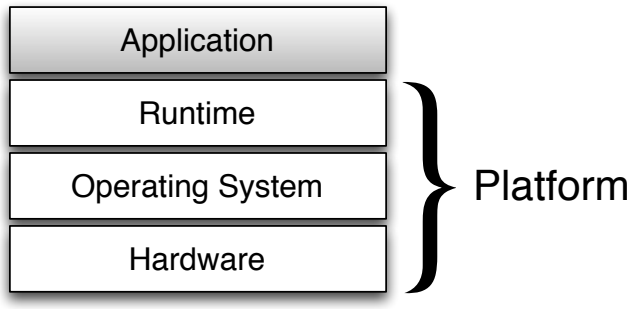


Figure 6: Platform Stack

A computing platform stack, as shown in Figure 6, typically consists of three layers. The first or the bottom-most layer is the *Device hardware*, which consists of the physical components of the system. The next layer, *Operating system* atop the device hardware, provides various hardware management functionality and other common services to the software that runs on the layers above it. The third layer is the *Application runtime*, which provides runtime support for software applications running on top of itself. Runtimes can vary from frameworks and libraries to interpreters or emulators. These three layers together constitute a computing platform and provide services to enable the application's execution. The *Software application*,

constructed by developers, uses these services and implements all the functionality required from it. While developing a multi-platform application, any differences in a layer of the computing platform should be handled by another layer above it or finally by the application itself. If such a difference is not handled suitably, it can lead to a user observable difference across the two platforms. Since, end users use multiple platforms to access such software, it is crucial to provide them with a consistent and reliable user experience across these platforms. Hence, these differences should be abstracted away in case of applications running across multiple platforms. For this purpose, developers mainly follow three approaches to develop the different multi-platform versions of the software, as described in the sections below.

2.2 The Single Web Approach

Web applications are popular means of delivering software. Developers can author a single web application and make it available to web browsers running on multiple platforms, which consist of a variety of hardware devices with different operating systems. This is made possible by standardization of web technologies implemented by all web browsers. As shown in Figure 7, the same web application can be interpreted by different web browsers on a variety of platforms. However, there can be subtle differences in web browsers across platforms, which can lead to a situation where web applications differ in look, feel, and functionality when run on different web browsers. We call such differences, which may range from minor cosmetic differences to crucial functional flaws, *cross-browser incompatibilities* (XBI). In the rest of this section, we describe details of web applications and the different reasons for cross-browser differences.

Web Applications Web applications are based on a client-server computing model. In a typical scenario, a human user interacts with the client-side of a web application through a web browser that runs on a computing device (e.g., a desktop PC). Users

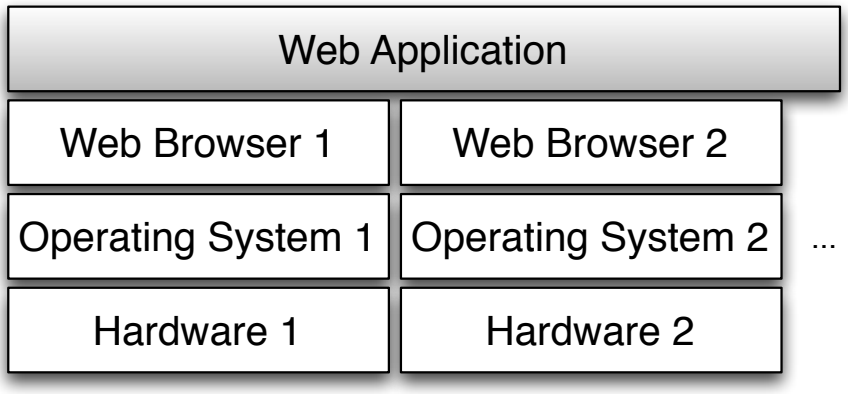


Figure 7: Single Web Application to target all platforms

view web pages, enter data, and perform actions, such as clicks on widgets (e.g., buttons or hyper-links). These interactions generate requests to the server, and the server responds to such requests with updates to the current web page, encoded in HTML (Hyper-Text Markup language) or XML (eXtensible Markup Language), and to other associated resources, such as style information in CSS (Cascading Style Sheets), client-side code (e.g., JavaScript), images, and so on. These resources are then used to compute and render an updated web page in the web browser. The recent trend is to handle an increasing portion of the user interactions entirely on the client side, using JavaScript code and other components, such as Flash, to compute responses and updates to the current web page. In fact, many of the web pages viewed by the user may have no corresponding REST-based [25] URI. This is typical of several modern web applications based on the AJAX paradigm.

The Web Browser: A Source of Cross-browser Differences Modern web browsers are fairly sophisticated applications comprised of a number of components. A typical architecture of a web browser is presented in [30]. Of the many functional components at work in a browser, there are three that are of specific interest for understanding the reasons for cross-browser incompatibilities. The first and most

important among them is the *layout engine*, which is responsible for rendering a web page by combining the structural information in the HTML for the page with the style information in CSS stylesheets. The browser also maintains a *DOM (Document Object Model)* representation of the page in memory to allow client-side scripts (e.g., JavaScript code) to modify the web page dynamically. The layout engine is the primary source of cross-browser differences, as the same HTML/DOM and CSS can produce different-looking pages in different browsers. The second component is the *event-processing engine*, or the *DOM engine*, which couples a user action, such as a mouse click on a specific location, with the execution of specific event-handling client-side code. This engine also performs changes in the DOM based on the DOM-API of the browsers. Browsers also differ in their event-handling algorithms, as well as in the DOM-API they support. This is another source of cross-browser differences. Thus, the same user action can produce a different change to the DOM. A third source of difference is the *JavaScript engine*—the runtime environment for executing JavaScript code within the browser. Subtle but definite differences exist between the JavaScript engines of different browsers, which result in differences in behavior. It is noteworthy that standards do exist for various client-side technologies, such as HTML, CSS, DOM, and ECMA-Script. However, browsers typically implement their own variants of these standards.

Due to these factors, the developer needs to test the application across different browsers to uncover and fix the different cross-browser incompatibilities. Later in Chapter 3, I present the X-PERT technique, which not only automates the detection of such issues but also assists the developer to fix them.

2.3 Mobile Web Applications

With the proliferation of mobile devices, an increasing number of people are using mobile devices to access web applications over the Internet. Although modern mobile

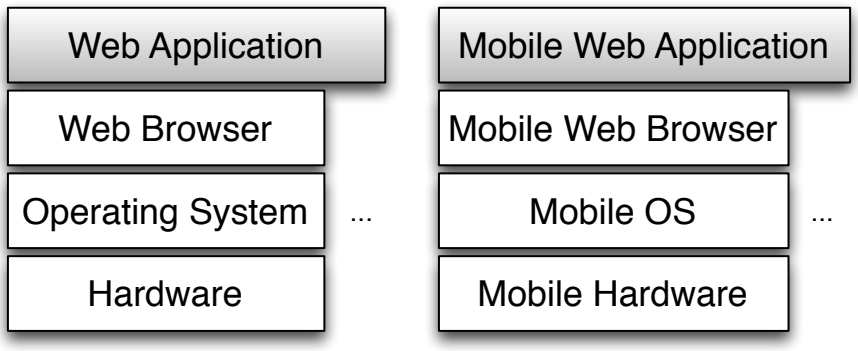


Figure 8: Separate Mobile Web Application for Mobile platforms

devices have a high degree of support for web technologies, they are still different from traditional computers in terms of form factor, user interface, unique hardware support, different contexts of operation, etc. Due to this, a rich web application written with desktop browsers in mind, might be inefficient and thus lead to a low user experience on mobile platforms. To address this issue, companies typically develop one or more customized mobile-web optimized version of the web application, which is different from the desktop version of the application, as shown in 8.

Most importantly, the user interface of a mobile web application is significantly different from the desktop application [41]. This difference can be observed in the widgets and the web design patterns used for these versions. The existence of these differences increases the complexity of the presentation layer, since the developer needs to maintain and support different versions for the desktop and mobile platforms. Typically the web developer creates different presentation layers of the web application and hosts them on the web server. Based on the web browser used, a suitable desktop or mobile view of the web application is served. However, typically both the desktop and mobile clients communicate with the same core server-side web application. In this approach, the standard web browser platform serves as the runtime running a different application version for the desktop and mobile platforms. This might lead to several issues because the developer will need to duplicate the

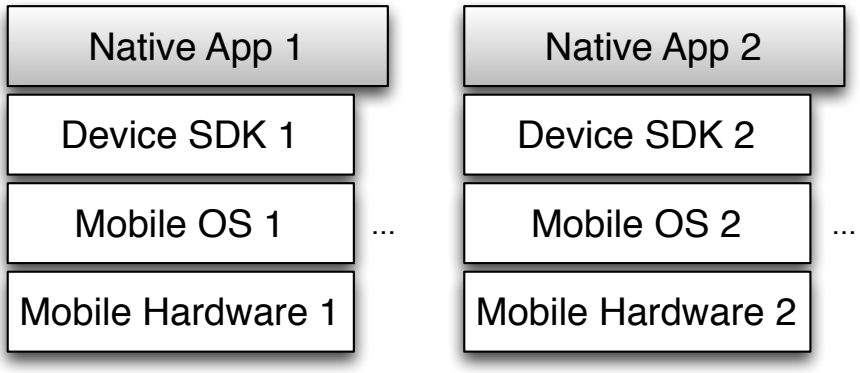


Figure 9: Native Mobile Application tailored for each Mobile platform

testing and maintenance activities for the desktop and mobile-web versions of the applications. Also, it is possible to have an inconsistency between the feature sets provided by the applications. In particular, features present in one version of the application, might not be present in the other or might be implemented in a very different, non-intuitive manner for the end users of the application. Such a situation can drastically affect the end user's experience on the platform as they would not be able to access those features on the affected platforms. Thus, to eliminate such situations, developers needs to be aware of these missing features, without a need for more manual work on her part. This will enable them to address these issues before the application is released and will also reduce potential support requests in future. In Chapter 4, I introduce the FMAP technique to match feature across desktop and mobile versions of web applications, and to report features that are missing across these versions.

2.4 Native Mobile Applications

Mobile devices support native applications, often called as *Apps*, which can take full advantage of the capabilities of the device, such as camera, GPS, accelerometer, compass, contact list etc. In addition, native applications can also incorporate touch gestures, notifications and offline capabilities, which allows applications to be more

interactive and connected to the user. Hence, many software companies develop such native applications to take advantage of such hardware level features. As shown in Figure 9, in this approach, the different platform layers are entirely different. The runtime in the case of native applications is a custom software development toolkit (SDK) for each platform. Unlike the web browser runtimes, as in earlier scenarios, this runtime does not have a common specification and is substantially different in architecture and design. Examples of such native mobile platforms include mobile phones running Android, iOS and Windows mobile operating systems, which include their own custom runtime libraries and SDKs. Not only are the native mobile platforms different, but the applications are also written using different programming languages and library components. Hence, a native application's versions developed for two mobile platforms are essentially totally different software applications offering similar functionality to its users at a high level.

Developers make this extra effort, of developing separate native apps, to make use of the different hardware level features exposed by the SDK APIs. This allows them to build optimized versions of the application for the particular platform. Not only does the developer need to create such applications, but they are also required to duplicate testing and maintenance tasks. This includes rewriting from scratch the test cases for every platform. Thus, any technique which aids such repetitive work through automation, would be helpful for developers of cross-platform native mobile applications. Hence, in my work, I present a preliminary technique, MigraTest in Chapter 5 to automatically translate test cases of the mobile application from one platform to the other.

2.5 Other Approaches

Other than the aforementioned three approaches, developers also follow hybrid approaches to ease development and to increase code reuse. A popular example of

such a hybrid approach is the development of native mobile application using web technologies, such as HTML, CSS and JavaScript. Apache Cordova[1] and Appcelerator Titanium[2] are two examples of frameworks that allow such development. These frameworks provide an extended web browser component, also known as a WebView, which provides access to the native mobile functionalities (e.g., accessing the file system, contacts, or device sensors) through a JavaScript API. The application written using web technologies is then compiled with this browser component for multiple platforms. The parts of the application written using web technologies is then visible through the WebView specific to the particular platform. Thus, this approach allows the distribution of the mobile application across different platforms.

Although this thesis does not directly address testing and maintenance problems in all such hybrid scenarios, the techniques presented can still be partially applied in such cases. For instance, the common part in hybrid mobile applications, built using web technologies, might present issues similar to Cross-browser issues described in Section 2.2. Thus, a technique for detecting such issues for web browsers can be customized and applied to detect inconsistencies introduced by the browser components on different platforms.

[1] Apache Cordova - `http://cordova.apache.org`
[2] Appcelerator Titanum - `http://www.appcelerator.com/titanium`

CHAPTER III

CROSS-BROWSER TESTING OF WEB APPLICATIONS

This chapter presents X-PERT, a new comprehensive technique and tool for detection of XBIs that addresses the limitation of existing approaches. First, X-PERTś approach is derived from an extensive and systematic study of real-world XBIs in a large number of web applications from a variety of different domains. Besides showing that a large percentage of web applications indeed suffer from XBIs (over 20%), thus providing further evidence of the relevance of the problem, the study also allowed us to identify and categorize the most prominent feature differences that a human user would most likely perceive as actual XBIs.

Second, X-PERT is designed to be a comprehensive and accurate framework for detecting XBIs. It integrates differencing techniques proposed in previous work with a novel technique for detecting layout errors, by far the most common class of XBIs observed in our case study (over 50% of web-sites with XBIs contained layout XBIs). This allows X-PERT to detect the entire gamut of XBI errors and do so with very high precision.

Finally, by targeting the most appropriate differencing technique to the right class of XBIs, X-PERT usually reports only one XBI per actual error, unlike other techniques (*e.g.,* [51,53]), which typically produce several duplicate error reports. For example, the movement of a single element on a page can have a domino effect on the positions of all elements below it. CROSSCHECK [51] might report all such elements as different XBIs, while our current approach would identify only the offending element. This improvement greatly simplifies the task of understanding XBIs for the developer.

The main contributions of this work are:

- A systematic study of a large number of real-world web applications that helps develop a deeper, realistic understanding of real-world XBIs and how to detect them.
- A comprehensive approach for XBI detection, called X-PERT, that integrates existing techniques with a novel approach to detecting layout XBIs in a single, unifying framework.
- An implementation of our X-PERT approach and a thorough empirical study whose results show that X-PERT is effective in detecting real-world XBIs, improves on the state of the art, and can support developers in understanding and (eventually) eliminating the causes of XBIs.
- A public release of our experimental infrastructure and artifacts (see `http://gatech.github.io/x-pert/`), which will allow other researchers and practitioners to benefit from them and build on this work.

3.1 Motivating Example

In this section, we introduce a simple web application that we use as a motivating example to illustrate different aspects of our approach. The application, referred to as **Conference** hereafter, is the web site for a generic conference.

Figure 10 provides an abstract view of this web site, as rendered in the Mozilla Firefox browser. The site consists of three interlinked dynamically generated pages that show the conference venue details, the key dates of the main conference activities, and the list of accepted papers. The buttons labeled **HOME**, **DATES**, and **PAPERS** can be used for navigating between different pages. Alternatively, the hyperlinks at the bottom of each page can also be used for navigation. The figure shows these inter-page transitions using two different kinds of edges, where dashed edges correspond to a button push, and solid edges correspond to a click on a link.

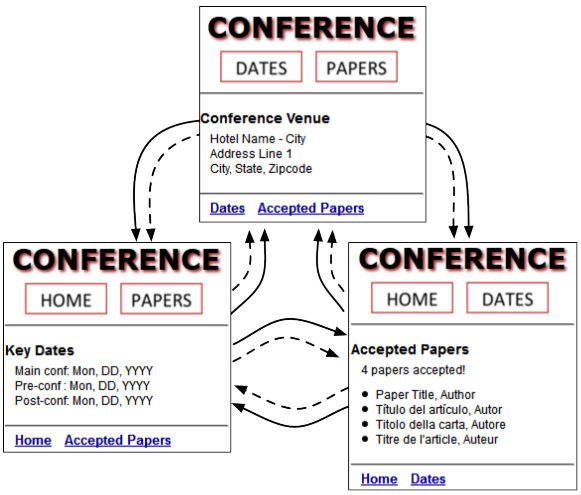
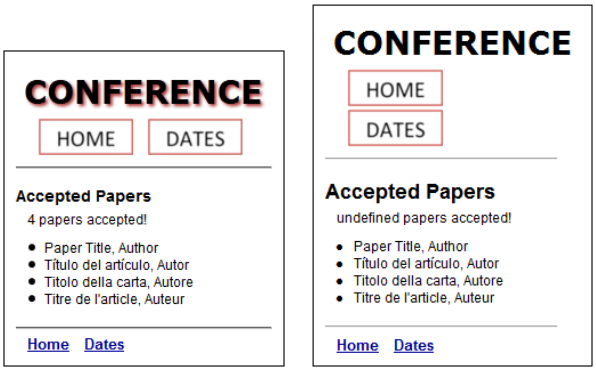


Figure 10: State Graph for web application Conference in Mozilla Firefox.



(a) Mozilla Firefox (b) Internet Explorer

Figure 11: One web page of Conference rendered in two browsers.

When rendered in Mozilla Firefox (FF) and Internet Explorer (IE), our example application manifests one behavioral and three visual XBIs (the latter shown in Figure 11). We discuss these XBIs and their causes individually.

XBI #1: Buttons HOME, DATES, and PAPERS do not produce any response when clicked in IE (*i.e.,* the dashed transitions in Figure 10 do not occur in IE), which prevents the users from accessing part of the application's functionality. The cause of this XBI is the following HTML code (shown for the DATES button only, as the other ones are analogous):

```
<img src="dates.png" onclick="navigate(event)" />
```

The application implements the buttons with `<img>` tags and associates the JavaScript event handler `navigate` to button-click events using the `onclick` attribute of such tags. Because IE does not support the `onclick` attribute for the `<img>` tag, the buttons are unresponsive in IE.

XBI #2: The buttons in the pages are arranged horizontally (left to right) in FF, but vertically in IE. The reason for this layout related XBI is that the application sets the total width of the button bar to 225 pixels. Due to differences in the default border and padding around button images in FF and IE, the second button does not fit in this width in IE and goes to the next line.

XBI #3: The number of accepted papers appears as `'undefined'` in IE. This issue is caused by the following JavaScript code:

```
var count = $("paperlist").childElementCount
                  + " papers accepted!";
$("papercount").innerHTML = count;
```

In the code, the list of papers is implemented as a list element (`<li>`) with id `''paperlist''`. The code uses property `childElementCount` to query the size of this list and adds it to the string that prints the number of papers. (We use `$("paperlist")` as a shorthand for the complete expression, which is `document.getElementById("paperlist")`.) Because the `childElementCount` property is not supported in IE, the query returns `'undefined'`, which results in the observed error.

XBI #4: The page title has a (red) shadow in FF and no shadow in IE. This last XBI is due to the following CSS property of the page title, which is an `<h1>` element:

```
h1{text-shadow: 2px 2px 2px red;}
```

Similar to the previous XBI, because the `text-shadow` property is not supported in IE, the shadow is absent in the IE rendering of the application pages.

3.2 Study of Real-World XBIs

As we discussed in the Introduction, the starting point of this work was a study of a large number of real-world XBIs. The goal was to provide a deeper understanding that could guide the re-targeting of existing XBI detection techniques and possibly the development of new ones.

In order to have an adequate sample of web applications for our study, we set the number of web sites to be studied to 100. Also, to avoid bias in the selection of the web sites, we selected them randomly using Yahoo!'s random URL service, available at `http://random.yahoo.com/bin/ryl`. For each web site selected, we followed the following process. First, we opened the web site using two different browsers: Mozilla Firefox and Internet Explorer. Second, a manual examination of the site was performed on the two browsers by studying both the visual rendering of the pages and their behavior when subjected to various stimuli. To limit the time requirements for the study, we selected a time limit of five minutes per site for the examination. This resulted in a total of over eight hours of manual examination, spread across several days. Finally, we analyzed the XBIs identified to categorize them based on their characteristics. We now discuss the finding of the study.

One striking result of our study is that the problem of XBI detection is quite relevant: among the 100 web sites examined, 23 manifested XBIs. This result is even more surprising if we consider that the examination involved only two browsers and a fairly limited observation time. More issues may appear if additional browsers and platforms, or a more extensive observation, were to be considered.

The study of the characteristics of the identified XBIs clearly showed three main types of XBIs: structure, content, and behavior. A finer grained analysis further allowed us to identify two subcategories for content XBIs: text and appearance. We describe these categories in detail below.

Table 1: Categorization of the real-world XBIs we found in our study.

Structure		13
Content	Text	5
	Visual	7
Behavior		2

- *Structure XBIs:* These XBIs manifest themselves as errors in the structure, or layout, of individual web pages. For example, a structure XBI may consist of differences in the way some components of a page (*e.g.,* widgets) are arranged on that page. XBI #2 in the example of Section 3.1 is an instance of such an XBI.
- *Content XBIs:* These XBIs involve differences in the content of individual components of the web page. A typical example of this type of XBIs would be a textual element that either contains different text when rendered in two different browsers or is displayed with a different style in the two cases. We further classify these XBIs as *text-content* or *visual-content* XBIs. The former category involves differences in the text value of an element, whereas the latter category refers to differences in the visual aspects of a single element (*e.g.,* differences in the content of an image or in the style of some text). XBIs #3 and #4 (Section 3.1) are instances of text-content and visual-content XBIs respectively.
- *Behavior XBIs:* These type of XBIs involve differences in the behavior of individual functional components of a page. An example of behavioral XBI would be a button that performs some action within one browser and a different action, or no action at all, in another browser. XBI #1 from Section 3.1 is a behavior XBI.

Table 1 shows, for each category of XBIs that we identified, the number of web sites in the study sample that exhibit that type of issue. Note that the sum of the values in the last column is higher than the total number of web sites with XBIs (23) because a single web site can contain multiple types of XBIs.

The above statistics, as well as a deeper analysis of each of the observed XBIs,

provided the following key insights:

1. The three categories of XBIs are independent, that is, there is typically little or no correlation between the occurrence of XBIs in one category and another.
2. The three categories of XBIs are qualitatively quite distinct. Intuitively, while behavior of a widget refers to how it respond to a user action, structure denotes where and how it is arranged on the page, and content refers to its appearance.
3. Structure XBIs are by far the most common category, occurring in 57% (13/23) of the subjects that had XBIs. Further, we observed that we tended to recognize a structure XBI through a difference in the relative position of an element with respect to its immediate neighbors, rather than a difference of its absolute size or position. (We hypothesize that most users will do the same.)

The first two insights suggest that the three categories of XBIs could be independently detected, and techniques specialized to each category should be used. This insight also partly explains why use of image-comparison techniques for detecting structure and content XBIs had a high false positive rate in our previous work [53]. The third insight motivated us to develop a novel approach for detecting structure XBIs based on the concept of *relative-layout comparison*. This technique is presented in Section 3.4. It also explained why using the absolute size or position of elements to detect structure XBIs in our previous work [51] resulted in many false positives.

3.3 Approach

Our overall framework for XBI detection falls into the broad category of "crawl-and-compare" approaches described in Section 6.1.4 and draws heavily on the findings of our case study in Section 3.2. The behavior capture step is fairly similar to the one used in [51]. However, the behavior comparison step, unlike [51] or any other previous work, is organized as a set of four independent and orthogonal algorithms, each

targeted to detect a specific category of XBIs: behavior, structure, visual-content, and text-content.

Further, the algorithms for behavior, visual-content, and text-content XBI detection are adapted from [51] but orchestrated differently and more effectively in the current work. The algorithm for detecting structure XBIs, which usually constitute the bulk of XBIs (see Table 1), is completely novel and a substantial improvement over previous work.

3.3.1 Terminology

Modern web applications are comprised of several static or dynamically generated web pages. Given a web page W and a web browser Br, $W(Br)$ is used to denote W as rendered in Br. Each web page is comprised of a number of web elements (*e.g.,* buttons, text elements) or containers of such elements (*e.g.,* tables). We use e to refer to an element of a web page. Further, each web page has a *DOM (Document Object Model)* representation, a layout, and a visual representation. We use D to refer to the DOM of a web page (or a portion thereof). The layout of a web page represents its visual structure. We model the layout as a set of potentially overlapping rectangles in a two dimensional plane and denote it as $\mathcal{L}$. Each rectangle represents an element of the page and is characterized by the coordinates (x_1, y_1) of its top-left corner and (x_2, y_2) of its bottom right corner. Thus, $\mathcal{L}(e) = ((x_1, y_1), (x_2, y_2))$ denotes the layout of element e. The visual representation of a web page is simply its two-dimensional image, as rendered within the web browser. Accordingly, the visual representation of an element is the image of the rectangle comprising the element.

3.3.2 Framework for XBI Detection

Algorithm 1 presents our overall approach for XBI detection. Its input is the URL of the opening page of the target web application, url, and the two browsers to be considered, Br_1 and Br_2. Its output is a list $\mathcal{X}$ of XBIs. The salient steps of this

Algorithm 1: X-PERT: Overall algorithm

Input : url: URL of target web application
Br_1, Br_2: Two browsers
Output: $\mathcal{X}$: List of XBIs

```
1  begin
2      X ← ∅
3      (M1, M2) ← genCrawlModel(url, Br1, Br2)
       // Compare State Graphs
4      (B, PageMatchList) ← diffStateGraphs(M1, M2)
5      addErrors(B, X)
6      foreach (S_i^1, S_i^2) ∈ PageMatchList do
           // Compare matched web-page pair
7          DomMatchList_i ← matchDOMs(S_i^1, S_i^2)
8          L_i^R ← diffRelativeLayouts(S_i^1, S_i^2, DomMatchList_i)
9          C_i^T ← diffTextContent(S_i^1, S_i^2, DomMatchList_i)
10         C_i^V ← diffVisualContent(S_i^1, S_i^2, DomMatchList_i)
11         addErrors(L_i^R, C_i^V, C_i^T, X);
12     return X
```

approach are explained in the following sections.

Crawling and Model capture: The first step is to crawl the web application, in an identical fashion, in each of the two browsers Br_1 and Br_2, and record the observed behavior as navigation models M_1 and M_2, respectively. The navigation model is comprised of a state graph representing the top-level structure of the navigation performed during the crawling, as well as the image, DOM, and layout information of each observed page. This is implemented by function *genCrawlModel*() at line 3 and is similar to the model capture step in CROSSCHECK [51].

Behavior XBI Detection: The next step is to check the state graphs of navigation models M_1 and M_2 for equivalence. This is done using the algorithm for checking isomorphism of labelled transition graphs proposed in [45]. Function *diffStateGraphs*() (line 4) performs this operation. This comparison produces a set of differences, $\mathcal{B}$, and a list *PageMatchList* of corresponding web-page pairs S_i^1, S_i^2 between M_1 and M_2. The differences in $\mathcal{B}$ are attributable to missing and/or mismatched inter-page transitions. Since these transitions characterize the dynamic behavior of the web application, $\mathcal{B}$ represents the behavior XBIs as detected by Algorithm 1. The algorithm

then iterates over the list of matched web-page pairs in *PageMatchList* and compares them in various ways to detect other kinds of XBIs (lines $6-13$).

DOM Matching: To compare two matched pages S_i^1 and S_i^2, the algorithm computes a list $DomMatchList_i$ of corresponding DOM element pairs in S_i^1 and S_i^2. This is implemented by function *matchDOMs*() (line 7) and done based on a *match index* metric for DOM element correspondence. This metric was first proposed in [53] and further developed in [51]. The match index uses a weighted combination of (1) the XPath (*i.e.,* path in the DOM—see `http://www.w3.org/TR/xpath/`), (2) DOM attributes, and (3) a hash of an element's descendants to compute a number between 0 and 1 that quantifies the similarity between two DOM elements. (See [51] for further details.) The computed $DomMatchList_i$ is used by several of the subsequent steps.

Structure XBI Detection: We introduce the notion of relative-layout comparison as the mechanism for detecting structure XBIs, which is one of the key contributions of this work. Function *diffRelativeLayouts*() (line 8 of Algorithm 1) compares pages S_i^1 and S_i^2 and extracts the set of relative-layout differences $\mathcal{L}_i^R$ that represent structure XBIs (also called relative-layout XBIs). The technique for detecting relative-layout XBIs is described in Section 3.4.

Text-content XBI Detection: These XBIs capture textual differences in page elements that contain text. To detect them, the text-value of an element is extracted from its DOM representation and compared with that of its corresponding element from $DomMatchList_i$. This operation is performed by *diffTextContent*() (line 9) and is similar to the method for extracting the $LDTD$ feature for machine learning in [51].

Visual-content XBI Detection: Visual-content XBIs represent differences in the visual appearance of *individual* page elements, such as differences in the styling of text or background of an element. To detect such errors, our approach takes the screen images of two corresponding elements and compares their color histograms using the χ^2 distance, similar to what we did in CrossCheck [51]. Unlike CrossCheck

however, which compared *all* DOM elements and generated many false positives, our new approach applies visual comparison only to leaf DOM elements, where it is most effective at detecting visual-content XBIs. Function *diffVisualContent*() (line 10) implements this operation. The XBIs extracted in this step and in the previous one are then added to the XBI list $\mathcal{X}$ (line 11).

3.4 Detecting Relative-Layout XBIs

Given a web page W and two different browsers Br_1 and Br_2, relative-layout XBIs represent discrepancies between the relative arrangements of elements on the layouts of $W(Br_1)$ and $W(Br_2)$. To accurately detect these issues, we introduce a formalism for modeling the relevant aspects of a page layout, called an *Alignment Graph*. To detect relative-layout XBIs, our approach performs the following two steps: (1) extract alignment graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ from the layouts of $W(Br_1)$ and $W(Br_2)$, respectively; and (2) compare $\mathcal{A}_1$ and $\mathcal{A}_2$ for equivalence and extract differences as relative-layout XBIs.

In the following sections, we formally define the alignment graph and the algorithms for extraction and equivalence checking of alignment graphs.

3.4.1 The Alignment Graph

The alignment graph is used to represent two kinds of relationships between the elements (rectangles) of the layout of a web page, namely, *parent-child* relationships and *sibling*. We introduce the relevant definitions for these two relationships.

Definition 1 (Contains Relation) *Given a set of elements D_s from a web page, a contains relation, $\prec$: $D_s \rightarrow D_s$, is defined between elements of D_s as follows. Given two elements $e_1, e_2 \in D_s$ with layout views $\mathcal{L}(e_1) = ((x_1^1, y_1^1), (x_2^1, y_2^1))$ and $\mathcal{L}(e_2) = ((x_1^2, y_1^2), (x_2^2, y_2^2))$ and XPaths $\mathcal{X}_1$ and $\mathcal{X}_2$, $e_1 \prec e_2$ if and only if*

- $x_1^1 \leq x_1^2 \wedge y_1^1 \leq y_1^2 \wedge x_2^1 \geq x_2^2 \wedge y_2^1 \geq y_2^2$ *and*

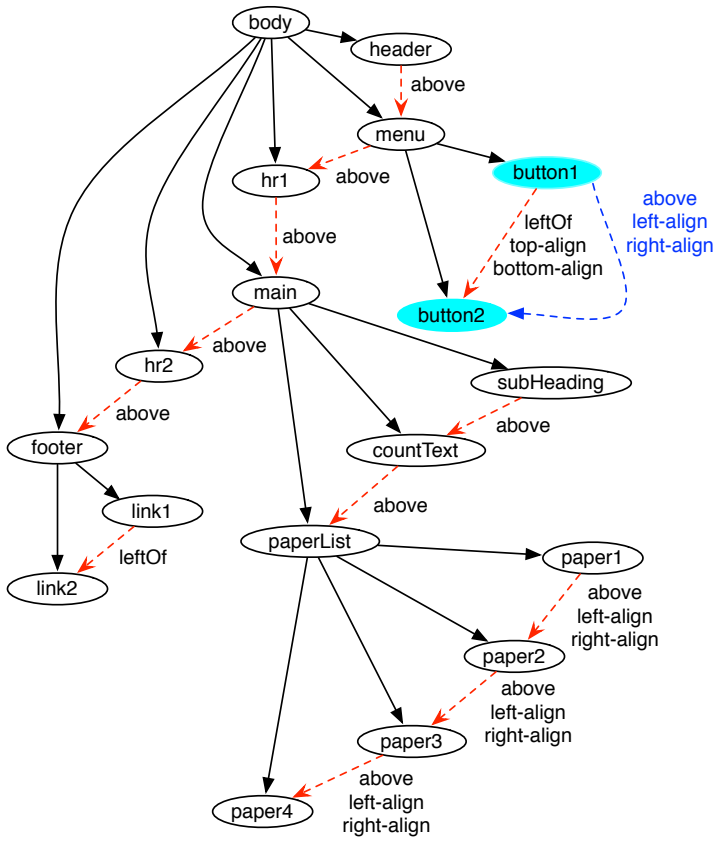


Figure 12: Alignment Graph for the web pages in Figure 11.

- *if* $\mathcal{L}(e_1) = \mathcal{L}(e_2)$ *then* $\mathcal{X}_1$ *is a prefix of* $\mathcal{X}_2$

Thus, a contains relation exists between e_1 and e_2 if either (1) rectangle $\mathcal{L}(e_2)$ is strictly contained within rectangle $\mathcal{L}(e_1)$ of e_1 or (2) e_1 is an ancestor of e_2 in the DOM, in the case where $\mathcal{L}(e_1)$ and $\mathcal{L}(e_2)$ are identical.

Definition 2 (Parent node) *Given a set of elements* D_s *from a web page, and two elements* $e_1, e_2 \in D_s$, e_1 *is a parent of* e_2 *if and only if* $e_1 \prec e_2$, *and there does not exist an element* $e_3 \in D_s$ *such that* $e_1 \prec e_3 \wedge e_3 \prec e_2$.

Thus, the parent of an element e is basically the "smallest" element containing e. Note that Definition 2 allows for elements to have multiple parents. However, to simplify the implementation, we use a simple metric (*i.e.*, the area) to associate each element with at most one parent.

Definition 3 (Sibling nodes) *Given a set of elements D_s from a web page. two elements $e_1, e_2 \in D_s$ are said to be siblings if and only if they have a common parent in D_s.*

Parent-child and sibling relationships can be further qualified with attributes specifying the relative position of the elements with respect to each other. For example, a child could be horizontally left-, right-, or center-justified and vertically top-, bottom-, or center-justified within its parent. Similarly, an element e_1 could be above, below, or to the left or right of its sibling element e_2. Further, e_1 and e_2 could be aligned with respect to their top, bottom, left, or right edges. These attributes can be simply computed by comparing the x and y coordinates of the elements in question.

Formally, an alignment graph $\mathcal{A}$ is a directed graph defined by the 5-tuple $(E, \mathcal{R}, \mathcal{T}, Q, \mathcal{F})$. Here, E is the set of vertices, one for each web page element. $\mathcal{R} \subseteq E \times E$ is a set of directed relationship edges, such that for elements $e_1, e_2 \in E$, there exists an edge (e_1, e_2) in $\mathcal{A}$ if and only if either e_1 is a parent of e_2, or e_1 and e_2 are siblings. (Although the sibling relation is symmetric, in practice only one of the edges (e_1, e_2) and (e_2, e_1) is sufficient to represent it, so we can arbitrarily choose one.) $\mathcal{T}$ is a set of the two types $\{parent, sibling\}$ for identifying an edge as a parent or a sibling edge. Q is a set of attributes (*e.g., left-align, center-align, above, leftOf*) used to positionally qualify the parent or sibling relationship. $\mathcal{F} : \mathcal{R} \mapsto \mathcal{T} \times 2^Q$ is a function that maps edges to their type and set of attributes.

Figure 12 shows the alignment graph for the web pages shown in Figure 11, where some sibling edges and edge attributes have been omitted to avoid cluttering. In the figure, parent edges are represented with black, solid lines, and sibling edges with red, dashed lines. Node labels indicate the element they represent. Nodes *button1* and *button2*, for instance, represent menu buttons HOME and DATES, respectively, and *header*, *footer*, and *main* represent the page header, footer, and the main content-bearing section (showing the accepted papers), respectively. The graph is identical for

Algorithm 2: ExtractAlignmentGraph

Input : W: Web page to analyze, Br: Web browser
Output: $\mathcal{A}$: Alignment Graph

```
1  begin
2      D ← extractDOM(W, Br)
3      L ← extractLayout(W, Br)
4      D_f ← filterDOM(D)
5      foreach e ∈ D_f do addNode(e, A)
6
7      addParentEdges(A, L, D_f)
8      addSiblingEdges(A)
9      foreach (v1, v2) ∈ parentEdges(A) do
10         addParentChildAttributes(A, L)
11     foreach (v1, v2) ∈ siblingEdges(A) do
12         addSiblingAttributes(A, L)
13     return A
```

the web pages in Figures 11a (FF) and 11b (IE), except for the sibling edge between the nodes *button1* and *button2*, which is represented as a dotted blue line for IE and as a red line for FF.

3.4.2 Extracting the Alignment Graph

Algorithm 2 describes our approach for extracting the Alignment Graph $\mathcal{A}$ of a target web page W with respect to a web browser Br. The algorithm first extracts the DOM D of $W(Br)$ (*extractDOM*(), line 2) and the layout $\mathcal{L}$ of $W(Br)$ (*extractLayout*(), line 3). Function *filterDOM*() then reduces D to D_f by pruning away DOM elements that have no bearing on the visible layout of the page (*e.g.,* `<a>`). Line 5 adds one vertex to $\mathcal{A}$ for each element in D_f. Layout $\mathcal{L}$ is then analyzed to deduce parent-child relationships between elements in D_f and insert parent edges between the corresponding vertices in $\mathcal{A}$. This is implemented by function *addParentEdges*() (line 6) and, similarly, for sibling edges by function *addSiblingEdges*() (line 7). The layout of each parent-child element pair is further analyzed to infer alignment attributes qualifying this relationship, which are then added to the relevant edge in $\mathcal{A}$ (lines $8-10$). This is similarly done for sibling edges through function *addSiblingAttributes*() (lines $11-13$).

Algorithm 3: addParentEdges

Input : A: Alignment Graph being built, $\mathcal{L}$: Layout of web page, D_f: Filtered DOM of web page

```
1  begin
2      E ← getListOfElements(D_f)
3      sort(E, g) // Sort E using compare function g
4      while size(E) > 1 do
5          e ← removeLastElement(E)
6          for index ← size(E) to 1 do
7              p ← getElement(E, index)
8              if contains(p, e, L) then
9                  insertParentEdge(A, p, e, L)
10                 break
```

Algorithm 3 computes the parent-child relationships among the nodes in D_f and inserts edges representing them into $\mathcal{A}$. First, the algorithm inserts the elements of D_f into a list E (function *getListOfElements*(), line 2). Then, list E is sorted using a compare function g (line 3) that satisfies the following property:

Property 1 *For a pair of elements $e_1, e_2 \in D_f$, if $e_1 \prec e_2$ then $g(e_1, e_2) = -1$.*

Finally, the algorithm iteratively removes the last element e from the sorted list E (line 5). It then scans E from left to right, while comparing e with each element, until it finds an element p such that $p \prec e$ (function *contains*(), line 8). From Property 1 of the sorted list E, p can be inferred to be the parent of e, so the algorithm adds a parent edge (p, e) to $\mathcal{A}$ (function *insertParentEdge*(), line 9).

It is fairly straightforward to prove that, given a compare function g satisfying Property 1, Algorithm 3 finds precisely one parent element, consistent with Definition 2, for each element in set D_f that has a parent, and adds a parent edge to $\mathcal{A}$ accordingly. Note that there are many possible compare functions g that satisfy Property 1. In our current implementation, we use a function that orders elements based on their geometric area and XPath.

3.4.3 Comparing Alignment Graphs

After extracting alignment graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ for $W(Br_1)$ and $W(Br_2)$, respectively, our technique checks the two graphs for equivalence; any difference found constitutes a relative-layout XBI. To do so, the technique uses the DOM matching approach we discussed in Section 3.3.2, which can determine corresponding elements in $W(Br_1)$ and $W(Br_2)$, and consequently, corresponding vertices in $\mathcal{A}_1$ and $\mathcal{A}_2$. Given a node $e \in \mathcal{A}_1$ (resp., $\mathcal{A}_2$), let $m(e)$ denote its corresponding node in $\mathcal{A}_2$ (resp., $\mathcal{A}_1$) as computed by our matching approach. Next, our technique iterates over each edge $r = (e_1, e_2)$ in $\mathcal{A}_1$ and checks that the corresponding edge $r' = (m(e_1), m(e_2))$ exists in $\mathcal{A}_2$. It further checks that these edges have identical labels, that is, $\mathcal{F}_1(r) \equiv \mathcal{F}_2(r')$. This check ensures that r and r' are of the same type and have an identical set of attributes. Any discrepancies in the edge correspondence is recorded as an error. The process is repeated in a similar way for $\mathcal{A}_2$. Each XBI is detected and reported in the form of differences in the "neighborhood" of a given element and its counterpart in the two alignment graphs. The neighborhood refers to the parent and siblings of the element, and to the edges between them. For example, in Figure 12, the comparison would yield a single XBI on *button1* caused by attribute differences between the *button*1 $\rightarrow$ *button*2 sibling edges in FF and IE. In FF, the edge indicates that (1) *button1* is to the left of *button2* and (2) the top and bottom edges of *button1* and *button2* are aligned. For IE, conversely, the dashed blue sibling edge indicates that *button1* is above *button2*, and the left and right edges of the two buttons are aligned. This is indeed the only layout XBI between the web pages in Figures 11a and 11b.

3.5 Implementation

We implemented our approach in a prototype tool called X-PERT (*Cross-Platform Error ReporTer*), which is implemented in Java and consists of three modules: *Model*

collector, *Model comparator*, and *Report generator* (Figure 13). Module *Model collector*, accepts a web application and extracts its navigation model from multiple browsers using an existing web crawler, CRAWLJAX [46]. CRAWLJAX acts as a driver and, by triggering actions on web page elements, is able to explore a finite state space of the web application and save the model as a state-graph representation. For each state (*i.e.,* page), *Model collector* also extracts an image of the entire page and geometrical information about the page elements by querying the DOM API.

Module *Model comparator* (MC) performs the checks needed to identify the different classes of XBIs defined in Section 3.2. First, the *Behavior checker* detects behavior XBIs by checking the state-graphs for the two browsers. Then, it passes the equivalent states from the two graphs to the *DOM matcher*, which matches corresponding DOM elements in these states. These matched elements are then checked for structural and content XBIs by the *Layout checker* and *Content checker*. The *Layout checker* implements the new relative-layout detection algorithm described in Section 3.4. Each element on the page is represented as a layout node, and its edge relationships are inferred using the geometric information captured earlier. To find differences in the neighborhood of matched nodes in two alignment graphs, X-PERT checks the nodes' incoming and outgoing edges, along with the corresponding edge attributes. Any discrepancy observed is attributed to the elements being compared. The *Content checker* compares both the textual and visual content of leaf DOM elements on the page. X-PERT performs textual comparison using string operations defined in the Apache Commons Lang library (`http://commons.apache.org/proper/commons-lang/`). To compare visual content, X-PERT uses the implementation of the χ^2 metric in the OpenCV computer vision toolkit [10].

Finally, the *Report generator* module generates an XBI report in HTML format, meant for the web developer, using the Apache Velocity library (`http://velocity.apache.org/`). These reports first present the behavioral XBIs, overlaid on a graph,

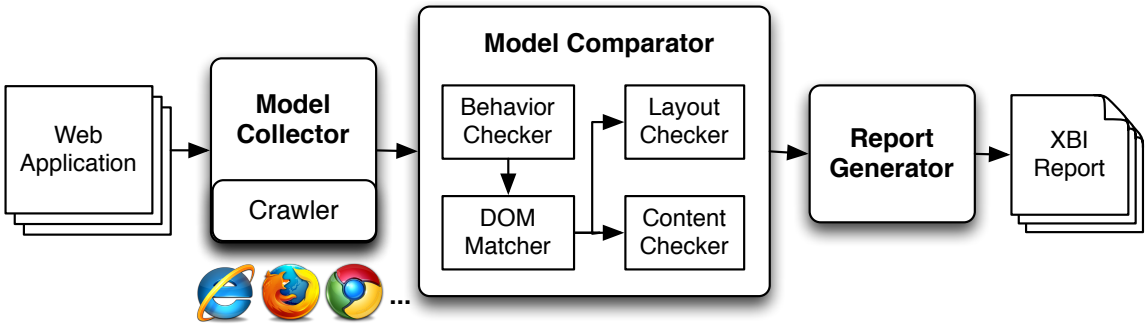


Figure 13: High-level architecture of X-PERT.

to depict missing transitions or states, if any. A list of XBIs is presented along with the pages where they appear. The user can select a particular page to see an instance of the XBI. These instances are identified using the XPaths and screen coordinates of the elements involved and also highlighted on two side-by-side screenshots of the affected page.

3.6 Empirical Evaluation

To assess the effectiveness of our technique for detecting XBIs, we used X-PERT to conduct a thorough empirical evaluation on a suite of live web applications. In our evaluation, we investigated the following research questions:

RQ1: Can X-PERT find XBIs in real web applications? If so, was the new relative-layout XBI detection algorithm effective in detecting the targeted issues?

RQ2: How does X-PERT's ability to identify XBIs compare to that of a state-of-the-art technique?

RQ3: How does X-PERT's screen-level XBI detection compare to that of a purely visual technique?

In the rest of this section, we present the subject programs we used for our evaluation, our experimental protocol, our results, and a discussion of these results.

Table 2: Details of the Subjects Used in X-PERT's Empirical Evaluation.

Name	*URL*	*Type*	*St.*	*Tr.*	DOM Nodes (per page)		
					max	*min*	*avg*
Organizer	http://localhost/organizer	Productivity	13	99	10001	27482	13051
GrantaBooks	http://grantabooks.com	Publisher	9	8	15625	37800	25852
DesignTrust	http://designtrust.org	Business	10	20	7772	26437	18694
DivineLife	http://sivanandaonline.org	Spiritual	10	9	9082	140611	49886
SaiBaba	http://shrisaibabasansthan.org	Religious	13	20	524	42606	12162
Breakaway	http://breakaway-adventures.com	Sport	19	18	8191	45148	13059
Conference	http://localhost/conference	Information	3	12	878	817	853
Fisherman	http://fishermanslodge.co.uk	Restaurant	15	17	39146	15720	21336
Valleyforge	http://valleyforgeinn.net	Lodge	4	12	5416	4733	5046
UniMelb	http://www.economics.unimelb.edu.au/ACT/	University	9	8	15142	12131	13792
Konqueror	http://www.konqueror.org	Software	5	4	17586	15468	16187
UBC	http://www.ubcsorority.com	Club	7	7	20610	7834	12094
BMVBS	http://m.bmvbs.de	Ministry	5	20	19490	12544	15695
StarWars	http://www.starwarsholidayspecial.com	Movie	10	9	28452	19719	22626

3.6.1 Subject Programs

Table 2 shows the fourteen subjects we used in our evaluation. Along with the name, URL, and type of each subject, the table reports the following information: number of states explored by X-PERT, number of transitions between these states, and minimum, maximum, and average number of DOM nodes analyzed per web page. (This latter information provides an indication of the complexity of the individual pages.)

The first six subjects (*i.e.,* Organizer, GrantaBooks, DesignTrust, DivineLife, SaiBaba, and Breakaway) had been previously used in the evaluation of CROSSCHECK. In addition, Organizer was also used for evaluating CROSSST [45]. Conference is our motivating example from Section 3.1 and was developed to show different classes of XBIs in a web application. The following three subjects—Fisherman, Valleyforge, and UniMelb—were obtained from the study of real world XBIs presented in Section 3.2. The main criteria for picking these subjects was the presence of known XBIs found in the study. All of the subjects mentioned so far had known XBIs, some of which were detected by previous techniques. To further generalize our evaluation, we selected four additional subjects using an online random URL service—`http://www.uroulette.com/`. (We used this alternative service because the

Yahoo! service we used in the study was discontinued.) These additional subjects are Konqueror, a web-based file manager, UBC, a student organization site, BMVBS, a mobile web application for the German ministry, and StarWars, a fan site.

3.6.2 Protocol

For our experiments, we set up X-PERT on a 64-bit Windows 7 machine with 4GB memory. X-PERT was configured to run two web browsers: the latest stable versions of Internet Explorer (v9.0.9) and Mozilla Firefox (v14.0.1). Our choice of these two browsers was due to their use in previous studies. In fact, the chosen browsers do not have any bearing on the technique and can be replaced with any browser of choice. Two subjects, Organizer and Conference, were hosted on a local Apache web server, whereas the remaining subjects were used live from their actual web sites. Note that we do not report any data on the performance of the tool because the whole analysis, including crawling, terminated in less than an hour.

To investigate RQ2, as a tool representative of the state of the art we selected CROSSCHECK [51]. Unlike X-PERT, CROSSCHECK does not combine XBIs across different web pages, thereby having the same XBIs possibly reported multiple times. Therefore, to perform a fair comparison, we implemented such a grouping on top of CROSSCHECK. We call this improved version CROSSCHECK+ in the rest of the chapter. The reports generated by these tools were manually inspected to find true and false positives. In addition, we manually analyzed the web pages analyzed by the tools to count all issues potentially detectable by a human user, which we use as an upper bound for the number of issues that a tool can detect. We use this number to calculate the recall of the results produced by the two tools.

3.6.3 Results

To answer RQ1, Table 3 presents a detailed view of X-PERT's results when run on the 14 subjects considered. The table shows, for each subject, the true and false

Table 3: X-PERT's Detailed Results from Empirical Evaluation.

Name	Behavior		Structure		Content				Total	
					Text		Image			
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
Organizer	1	0	9	0	0	0	0	0	10	0
GrantaBooks	16	0	11	0	0	0	0	0	27	0
DesignTrust	2	0	5	3	0	0	0	0	7	3
DivineLife	7	0	3	6	1	0	0	0	11	6
SaiBaba	2	0	2	9	0	0	0	0	4	9
Breakaway	0	0	10	2	0	0	0	0	10	2
Conference	2	0	3	0	1	0	1	0	7	0
Fisherman	1	0	3	1	0	1	1	0	5	2
Valleyforge	0	0	2	2	0	0	1	0	3	2
UniMelb	2	0	0	0	0	0	0	1	2	1
Konqueror	0	0	0	0	0	0	0	6	0	6
UBC	0	0	0	0	0	0	0	0	0	0
BMVBS	0	0	0	0	0	0	0	0	0	0
StarWars	0	0	12	0	0	0	0	0	12	0
TOTAL	33	0	60	23	2	1	3	7	98	31

Table 4: X-PERT's Results Compared to those of a State-Of-The-Art Technique.

Name	XBI	X-pert					CrossCheck+				
		TP	FP	Precision	Recall	Duplicate	TP	FP	Precision	Recall	Duplicate
Organizer	10	10	0	100%	100%	0	8	2	80%	80%	13
GrantaBooks	27	27	0	100%	100%	0	27	1	96%	100%	0
DesignTrust	7	7	3	70%	100%	0	6	122	5%	86%	3
DivineLife	11	11	6	65%	100%	0	10	24	29%	91%	3
SaiBaba	5	4	9	31%	80%	0	4	53	7%	80%	10
Breakaway	13	10	2	83%	77%	1	7	49	13%	54%	12
Conference	7	7	0	100%	100%	0	7	0	100%	100%	0
Fisherman	5	5	2	71%	100%	0	4	5	44%	80%	8
Valleyforge	3	3	2	60%	100%	0	1	1	50%	33%	0
UniMelb	2	2	1	67%	100%	0	2	27	7%	100%	0
Konqueror	0	0	6	—	—	0	0	11	—	—	0
UBC	0	0	0	—	—	0	0	1	—	—	0
BMVBS	1	0	0	100%	0%	0	0	2	0%	0%	0
StarWars	12	12	0	100%	100%	0	10	91	10%	83%	3
TOTAL	103	98	31	76%	95%	1	86	389	18%	83%	52

positives reported by X-PERT for each of the four types of XBI we identified, along with an aggregate total. As the results show, X-PERT reported 98 true XBIs and 31 false positives (76% precision). The detected issues included all four types of XBIs, with a prevalence of structure XBIs (60), followed by behavior (33) and content (5) XBIs. Based on these results, we can answer RQ1 and conclude that, for the subjects considered, X-PERT was indeed effective in finding XBIs. We can also observe that the new relative-layout XBI detection algorithm was able catch most of the issues in our subjects.

Table 4 summarizes and compares the results of X-PERT and CrossCheck+,

which allows us to answer RQ2. The table shows, for each subject, its name, the number of XBIs found by manual analysis (XBI), and the results of the two tools in terms of true positives (TP), false positives (FP), precision (Precision), recall (Recall), and duplicate reports (Duplicate) produced. As the table shows, X-PERT outperformed CROSSCHECK+ in terms of both precision and recall for all of the subjects considered, and often by a considerable margin. For subject DesignTrust, for instance, X-PERT produced 3 false positives, as compared to 122 false positives produced by CROSSCHECK+. On average, the precision and recall of X-PERT's results were 76% and 95%, respectively, against 18% and 83% for CROSSCHECK+. Our results also show that the X-PERT reported a negligible number of duplicate XBIs—only 1 versus the 52 duplicate XBIs CROSSCHECK+ reported. We can therefore answer RQ2 and conclude that, for the cases considered, X-PERT does improve over the state of the art.

3.7 Discussion

As our empirical results show, X-PERT provided better results than a state-of-the-art tool. We attribute this improvement, in large part, to our novel relative-layout detection technique. From our study of real world XBIs, presented in Section 3.2, it was clear that layout XBIs are the most common class of XBIs. In previous approaches, such as WEBDIFF or CROSSCHECK, these XBIs were detected indirectly, by measuring side-effects of layout perturbations, such as changes in the visual appearance or in the absolute size or position of elements. However, as demonstrated by our results, detecting side effects is unreliable and may result in a significant reduction in precision. In addition, a single XBI can have multiple side effects, which when detected by previous techniques would result in duplicate error reports.

One solution for eliminating duplication, used in previous techniques, is to cluster related XBIs. However, clustering can be imperfect, thereby including unrelated

issues in one cluster or separating related issues across multiple clusters. Moreover, developers still need to manually sift through the errors in a cluster to find the underlying cause of the XBI and related side effects. To alleviate this problem, X-PERT focuses each differencing technique (*i.e.,* visual comparison, text comparison, and layout differencing) where it can be most effective at detecting XBIs. By focusing the techniques on very specific problems, each XBI can be detected in terms of its principal cause, rather its side effects, which can be used to provide a better explanation of the XBI to the developers. In addition, we observed that such focused orchestration can detect more errors, which explains the improvement in the recall of the overall approach.

Another potential advantage of X-PERT is that it separates the individual techniques into different components, unlike previous approaches. Although we did not demonstrate this aspect in our study, intuitively this separation could allow developers to tune each of these components based on the kind of web application under test. For instance, developers could selectively use the behavioral detector, if such issues are more common in their web applications, or could turn it off to focus on other kinds of XBIs.

3.7.1 Threats to Validity

As with most empirical studies, there are some threats to the validity of our results. In terms of external validity, in particular, our results might not generalize to other web applications and XBIs. To minimize this threat, in our study, we used a mix of randomly selected real-world web applications and applications used in previous studies. The specific browsers used in the evaluation should not have affected the results, as our technique does not rely on browser specific logic and operates on DOM representations, which are generally available. Thus, we expect the technique to perform similarly on other browsers.

Threats to construct validity might be due to implementation errors in X-PERT and in the underlying infrastructure—especially with respect to the integration with the browser to extract DOM data. We mitigated this threat through extensive manual inspection of our results.

CHAPTER IV

DETECTING MISSING FEATURES IN A MULTI-PLATFORM WEB APPLICATION

This chapter presents FMAP, which is an automated technique for matching features across different versions of a multi-platform web application. The goal of this technique is to accurately identifying matching features across the desktop and mobile versions of a web application.We defined our technique based on the intuition that, although the front-ends of these platform-specific versions may look substantially different, in most cases they rely on the same back-end functionality. Specifically, if the platform-specific customizations are typically restricted to the client tier, with the server tiers mostly unchanged, exercising the same feature on two different platforms should generate largely similar communications between client and server in the two cases. Our technique therefore identifies and matches the features of a multi-platform web application by analyzing the client-server communication that occur when the application is used on the different platforms. At a high level, our technique operates in four main steps: (1) record traces of the network communication between the client and server of platform-specific versions of a web application, (2) model each trace as a sequence of basic actions, (3) identify a subset of these traces as feature instantiations, and (4) match the feature sets identified for each platform-specific version of the web application to identify matching and missing features across versions.

The main contributions of this work are:

- The introduction and definition of the notion of consistency between different, platform-specific versions of a web application.
- The definition of a technique for performing cross-platform feature matching for

web applications.

- The development of FMAP, a prototype tool that implements our technique and is publicly available, together with our experimental infrastructure (`http://gatech.github.io/fmap`).
- An empirical evaluation of our technique on nine real-world multi-platform web applications.

4.1 Motivating Example

In this section, we introduce a simple web application and use it as our motivating example to illustrate the challenges and opportunities for matching features across different platforms. The example web application *MakeMyPost.com*, as shown in Figure 14, is a content management system, and provides different front-ends for the desktop and mobile platforms. The first row shows two screens for the desktop version of the application and the second row shows three mobile screens.

When the user first loads the web application, she is taken to the login screen. The desktop and mobile versions of this screen have differences in their presentation as well as function. For example, the widgets for the login button and the alignment of the text box and the corresponding labels is different. Further, the "Remember me" check-box and the "Forgot pass" button, and their corresponding functionality, is not provided on the mobile view.

After login, the user is taken to the "Home" screen, where she can create a new post on the website. As shown, even this view is somewhat different on both platforms. Firstly, the navigation tabs present on the desktop version has been replaced by a dropdown on mobile. This dropdown can be seen in action on the third mobile screen. Although the functionality is preserved on the mobile web app, instead of clicking on the tabs, the user would select a dropdown option to navigate to the other screens of the web application. Secondly, the radio buttons, which allow the user on the

Figure 14: MakeMyPost.com Web Application for Desktop and Mobile Browsers

desktop to choose the "Post Type", are missing on the mobile screen. Finally, like the previous screen, the buttons use a different widget on both screens and appear different.

Thus, although the core functionality of the desktop web app is substantially mirrored in the mobile version there are significant differences in the style of widgets, the layout of various screens, and on occasion, the actions required to access specific functions. Thus, techniques based on comparing presentation-level information, such as screen layout and attributes of widgets would not work in this context. Tools for cross-browser compatibility checking [52] are once such example.

```
1. REQUEST: GET /index.php
2. RESPOSE: 200 OK, 'text/html'

3. REQUEST: GET /style.css
4. REPONSE: 200 OK, 'text/css'

5. REQUEST: GET /logo.png
6. REPONSE: 200 OK, 'image/png'

7. REQUEST: GET /script.js
8. REPONSE: 200 OK, 'text/javascript'

9. REQUEST: POST /login.php
       user=user1&pass=..&sid=w2s31
10. RESPONSE: 200 OK, 'text/html'

    ....
11. REQUEST: POST /create_blog.php
       title=..&content=..
12. RESPONSE: 200 OK, 'text/html'
```

(a) Desktop Trace

```
1. REQUEST: GET /index.php
2. RESPOSE: 200 OK, 'text/html'

3. REQUEST: GET /mobile_style.css
4. REPONSE: 200 OK, 'text/css'

5. REQUEST: GET /logo_small.png
6. REPONSE: 200 OK, 'image/png'

7. REQUEST: GET /mobile_script.js
8. REPONSE: 200 OK, 'text/javascript'

9. REQUEST: POST /login.php
       user=myUser&pass=..&sid=d4sW2
10. RESPONSE: 200 OK, 'text/html'

    ....
11. REQUEST: POST /create_blog.php
       title=..&content=..
12. RESPONSE: 200 OK, 'text/html'
```

(b) Mobile Trace

Figure 15: Network trace from MakeMyPost.com on desktop and mobile.

Now, let us consider the client-server communication originating from both application versions. We will consider a use-case where the user creates a post. For this purpose, first the user authenticates herself to the system from the login screen. Then she navigates to the home screen where she submits the post content. The corresponding network requests for this use-case are shown in Figures 15a and 15b, for the desktop and mobile platforms respectively. As one can notice, the requests made by both versions of the applications are largely similar, albeit with some minor differences. The first difference is in the requests to client-side scripts and styling information is different (i.e., the requests on lines 3–8, point to separate resources). Secondly, the requests made to the server-side scripts have differences in the submitted form data provided by the user as well as that generated by the application (e.g., the `user` and `sid` fields on line 9). However, the requests on lines 9 and 11, which invoke the "login" and "create_blog" functionalities on the server side respectively,

when taken together uniquely characterize the use-case shown in the example. These requests in fact correspond to the same action on either platform.

The intuition behind our approach, developed in this work, is that by analyzing use-cases in terms of the network traces they generate we can abstract away the irrelevant parts of the trace, *e.g.,* the user data. Further, by using the key actions that characterize these abstract use-cases we can successfully establish correspondence between different implementations of the same use-case on different platforms.

4.2 Terminology and Problem Definition

In this section, we define the terminology used for developing our approach in the next section. The terms are defined specifically in the context of the network level communication between the client and the server sides of web applications. They may carry different meanings in other contexts.

Definition 4 (Service) *A service is an atomic functionality offered by the web server to all clients, which may be invoked, potentially under different contexts, by different clients.*

In the *MakeMyPost.com* example, two services offered by the server are the login and create post functionalities.

Definition 5 (Request) *A request is a call made from the client browser to the server, to request display resources, exercise a service, or to navigate the user-interface to gain access to a particular service.*

Definition 6 (Response) *A response is the reaction of the server to a request from the client.*

Definition 7 (Trace) *A trace is an ordered sequence of requests and responses that is generated as a user exercises a given use-case on the application, through the client browser.*

Figure 15a shows a trace from the desktop version of *MakeMyPost.com*, corresponding to the use-case of logging in and creating a post. Figure 15b shows the corresponding trace for the mobile version. In this example, each of the traces contain 6 requests and 6 responses, as indicated. Note that only the requests corresponding to lines 9 and 11 invoke services (login and create post respectively), while the others request display resources or navigate the user-interface.

Definition 8 (Feature) *A feature is the functionality exercised by executing a specific set of services, provided by the web application, in a specific order.*

A feature can be exercised through any of several use-cases of the application, each of which exercise the services defined by the feature in the said order. Thus, a feature is, in effect, an *abstract* use-case, describing this set of concrete use-cases. The traces shown in Figure 15 exercise the features of logging in, followed by creating a post. Other variations of this use-case, interleaved with arbitrary navigation actions on the UI would correspond to the same feature, as would use-cases creating multiple posts. However, a use-case for logging in and simply browsing blog-posts, without creating a new one, would map to a different feature (since it does not exercise the service for creating a post).

Definition 9 (Action) *An action is a request with the user data and platform-specific resource references abstracted away.*

Thus, an action is essentially an abstract request. For our motivating example, the login request (line 9) can be made from different platforms, in different traces, and with different usernames and passwords. However, all such distinct requests access the same login service of the web application, on the server. Hence, all these requests correspond to the same action.

Definition 10 (Feature Equivalence) *Two application features, each from a different platform, are said to be equivalent if they correspond to exercising the same set of services on the server side and in the same sequence.*

Thus, the two traces shown in Figure 15 instantiate the equivalent "login and create blog" feature on the desktop and mobile platforms respectively. We would like to automatically establish such an equivalence across all the features available on each platform.

Given a web application with two versions $\mathcal{W}_1$ and $\mathcal{W}_2$, as implemented on two platforms, $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, we would like to establish a mapping of features between $\mathcal{W}_1$ and $\mathcal{W}_2$. As a starting point for analyzing the user-interfaces (UI) of $\mathcal{W}_1$ and $\mathcal{W}_2$ we assume that we are given sets of traces $\mathcal{T}_1$ and $\mathcal{T}_2$ generated from $\mathcal{W}_1$ and $\mathcal{W}_2$ respectively. These traces should exercise the features available on the respective interfaces. However, there are no other assumptions on trace sets $\mathcal{T}_1$ and $\mathcal{T}_2$. For example, $\mathcal{T}_1$ and $\mathcal{T}_2$ need not be minimal sets or correspond to each other in any way. In fact the trace sets need not even represent all the features of each UI. Our technique simply matches the features represented in the trace sets. These traces could be drawn from a variety of sources, such as from user-session data, from test-cases written for each application version or even by systematically crawling each web application [46]. Our technique makes no assumption regarding the sources of these traces either. Based on this, we can formally pose the *feature matching problem* as follows.

Definition 11 (Feature Mapping Problem) *Given two versions $\mathcal{W}_1$ and $\mathcal{W}_2$ of a web application, as implemented on two different platforms, and two sets of traces $\mathcal{T}_1$ and $\mathcal{T}_2$ drawn from $\mathcal{W}_1$ and $\mathcal{W}_2$ respectively, the feature mapping problem is to identify sets of features $\mathcal{F}_1$ and $\mathcal{F}_2$ represented in traces $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively, and a one-to-one relation $\mathcal{M} \subseteq \mathcal{F}_1 \times \mathcal{F}_2$, such that for any features $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$, $(f_1, f_2) \in \mathcal{M}$ iff features f_1 and f_2 are equivalent.*

The feature mapping problem, as posed above, presents the following challenges:

- **Action Recognition:** Although, each of the requests contained in the raw traces (trace sets $\mathcal{T}_1$ and $\mathcal{T}_2$) appear distinct, they are in fact instances of a small set of actions available on the UI of the web application. Thus, requests need to be appropriately abstracted and recognized as the appropriate action.
- **Trace Set Canonicalization:** Since we make no assumptions on the traces present in the provided trace-sets, it is quite conceivable that the trace-sets contain several traces representing a given feature. Thus, the trace-sets need to be canonicalized into a minimal set with precisely one representative for each feature.
- **Feature Mapping:** The minimal trace-sets obtained in the previous stage need to be mined for features which need to be mapped. Note that the requests (or actions) do not directly specify whether they represent a call to navigate the UI, procure presentation resources or actually exercise a service. Thus, the identification of service invokations and hence identification of features needs to be performed indirectly leveraging other information.

Our technique, developed in Section 4.3, presents our solution to these challenges.

4.3 Technique

In this section we develop our technique for accurately identifying matched and unmatched features across mobile and desktop versions of a web application. As stated in Section 4.2, we use a set of traces derived from client-server communication of each version as the basis for performing this matching. In our view, this interface is most appropriate for this task because it naturally abstracts away a lot of presentation-level differences, while preserving the functional structure of the use-case. Further, it allows us to develop our solution as a black-box technique, which is much easier to deploy and maintain than, for example, a (hypothetical) white-box technique based

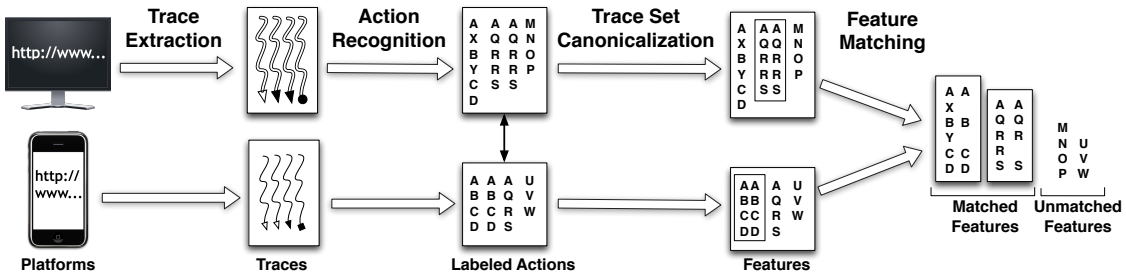


Figure 16: High-level overview of FMAP.

on analysis of server-side artifacts. Also, the use of traces is well suited to our application since the features we are attempting to compare are in fact abstract traces. Thus, more elaborate representations of the client user-interface, such as finite-state machine models [46, 56] or event-based models [43] would not be particularly useful in this context.

Figure 16 presents a high-level view of our technique, FMAP. The *first* step of FMAP is to collect a set of network-level traces from the two web application versions. These trace-sets form the basis of the subsequent feature mapping. The core feature mapping is largely independent of this trace collection. It consists of three principal steps, mirroring the three challenges discussed in Section 4.2. In the *first* step, the network traces are mined to identify requests which are instances of the same action. In this phase, all requests are abstracted and mapped onto a small alphabet of actions. In the *next* step, the abstract traces from each platform are clustered and canonicalized into a core set of traces with precisely one representative for each potential feature supported on that platform. In the *final* step, the canonicalized traces from the desktop and mobile platforms are compared against each other to find a correspondence between features. The matching from this step produces two results: (1) the mapping between the matched features of the application across the two platforms, and (2) the features which did not match and are possibly missing in the desktop or mobile version of the web application.

In the remainder of this section, we will explain the details of each of these steps

of our technique using our motivating example presented in Section 3.1.

4.3.1 Trace Extraction

The goal of this step is to automatically capture network level traces of the web application from both desktop and mobile platforms. A trace is captured as a user is interacting with the web application and performing meaningful actions to access the features offered by the web application. For every use case, which the user exercises, the technique captures the request-response pairs sent by the browser along with certain meta-data related to each pair. In particular, for HTTP requests, the technique collects and saves the URL path (*path*) and request parameters (*params*). The latter contains the information sent in the request as a key-value pair. For each HTTP response header, the technique saves the *response code* and the *MIME type* of the resource returned. The response code contains the status of the response, which can be indicative of either success, redirection or error. This response information is used in the next step to determine how the request information should be used for recognizing actions. Figure 15 contains several examples of such request-response pairs.

4.3.2 Action Recognition

The goal of this step is to identify intrinsically similar requests, appearing in different network traces, and recognize them as instances of the same action. As described in Algorithm 4, *RecognizeActions* is the main function in this algorithm. It takes a set of network traces from the two platforms, and returns a set of labeled actions. The key functions involved in this step are: 1) Trace simplification ($TraceSimplify$), to convert traces into sequences of keyword sets, 2) Action clustering ($ClusterActions$), to cluster related requests into the same action, and 3) Action canonicalization, to assign same symbols to nearly similar actions across different platforms. These steps are explained in detail below:

Algorithm 4: Action Recognition

```
/* RecognizeActions */
Input  : 𝕋_d, 𝕋_m: Set of traces from desktop and mobile
Output: 𝔸_d, 𝔸_m: Set of labeled actions for desktop and mobile
begin
    C_d ← ClusterActions(𝕋_d)
    C_m ← ClusterActions(𝕋_m)
    // Action Mapping
    Map ← { }
    foreach c1 ∈ C_d do
        foreach c2 ∈ C_m do
            if isSimilar(c1, c2) then
                if c1 ∈ Map or c2 ∈ Map then
                    c1 ← c1 ∪ Map.remove(c1)
                    c2 ← c2 ∪ Map.remove(c2)
                Map.add(c1 ↦ c2)
    𝔸_d ← [], 𝔸_m ← []
    foreach (c1, c2) ∈ Map do
        action ← getNewSymbol()
        𝔸_d.assign(c1, action)
        𝔸_m.assign(c2, action)
    foreach c1 ∈ C_d and c1.action == null do
        𝔸_d.assign(c1, getNewSymbol())
    foreach c2 ∈ C_m and c2.action == null do
        𝔸_m.assign(c2, getNewSymbol())
    return 𝔸_d, 𝔸_m
```

```
/* ClusterActions */
Input  : 𝕋: Set of traces
Output: ℂ: Cluster of actions
begin
    𝒦 ← TraceSimplify(𝕋)
    // Level 1 Clustering
    L1Cluster ← SimpleCluster(𝕋, url_path_equals)
    // Level 2 Clustering
    L2Cluster ← { }
    JD ← {JaccardDistance(k_1, k_2) | k_1, k_2 ∈ 𝒦}
    underCluster ← split(L1Cluster, size == 1)
    overCluster ← split(L1Cluster, size > 1)
    L2Cluster.add(AggloCluster(underCluster, JD, (<, t_1)))
    foreach c ∈ overCluster do
        L2Cluster.add(AggloCluster(c, JD, (>, t_2)))
    return L2Cluster
```

```
/* TraceSimplify */
Input  : 𝕋: Set of network traces = {𝒯_1, 𝒯_2, .., 𝒯_n}
Output: 𝒦: Set of keyword tuple sequences = {k_1, k_2, .., k_m}
begin
    𝒦 ← ( )
    foreach 𝒯 ∈ 𝕋 do
        foreach ⟨request, response⟩ ∈ 𝒯 do
            while isRedirect(response.code) do
                response ← followRedirect(response)
            if isCodeOrData(response.type) then
                k ← getKeyws(request.path, request.qs)
                 𝒦.add(k)
    return 𝒦
```

4.3.2.1 Trace Simplification

The goal of this step is to extract a set of keywords from each request, which are later used to group similar requests. As shown in the algorithm (lines 33 –42), the $TraceSimplify$ function takes a set of network traces and returns a set of keyword tuple sequences, each corresponding to a provided trace. To achieve its goal, the technique first, removes redundant requests occurring due to HTTP redirection and assigns the *MIME type* of the final resource to the originating request (lines 37 – 38). This *MIME type* is used by the function call $isCodeOrData$ to only consider requests related to client-side code or data resources (line 39). All reqyests to resources related to style or binary files are hence ignored at this step. This is essential since the technique aims to abstract out information relating to the visual rendering of the page. For our motivating example (Figure 15), this step ignores requests on lines 3 and 5 for both platforms.

Next, the technique extracts all the words present in the request URL path and request parameters of these resources (line 40). Our notion of a word is a sequence of alphabets separated by the reserved URL characters [9]. This allows us to ignore numeric values as well as randomly generated tokens or session identifiers. We also ignore the words belonging to a list of known file extensions [37]. The extracted words are further simplified by converting them to their lemmas by using Lemmatisation [42]. This process converts different suffixed or prefixed forms of the same word into one, thereby making them standard across different occurrences. At the end of this step, the technique has a sequence of keyword tuples for each trace. For example, the sequence corresponding to the desktop trace of our example application is `[('index'), ('script'), ('login', 'user', 'pass', 'sid'), ('create', 'blog', 'title', 'content')]`.

4.3.2.2 Action Clustering

This step is used to map intrinsically similar requests onto the same action. This is done by performing a two-level clustering as shown in the *ClusterActions* routine. Assuming a blackbox view of the server-side from the client, the URL path is used to indicate the service which is invoked. Thus, the first level of clustering combines all requests made from one platform with the same URL path into the same cluster. The *SimpleCluster* routine (line 24) takes the traces and uses this URL equality notion to cluster the requests. After this clustering, another level of clustering is needed to refine the clusters based on other URL parameters.

The second level of clustering, as shown on (lines 25–32), is used to further refine two classes of clusters obtained from the first level of clustering: 1) Over-clustered requests, which result from different requests being clustered together, and 2) Under-clustered requests, which are similar requests put into separate clusters. A practical case of over-clustered requests is when a request parameter is used reflectively to determine the server-side function to be invoked. Under-clustered requests can be illustrated by two requests invoking the same service, but whose URL path contains dynamic fields possibly entered by the user or generated by the application. For this step, we use agglomerative clustering [42], which is a kind of hierarchical clustering that uses a distance metric to iteratively merge two items by varying the threshold on the distance metric. We use the Jaccard distance metric [33] for this step, which is defined as:

$$JaccardDistance(a, b) = 1 - \frac{|words(a) \cap words(b)|}{|words(a) \cup words(b)|}$$

Here, (a, b) are two requests and $words(a), words(b)$ are the respective set of keywords computed in the trace simplification step. The Jaccard Distance measures the dissimilarity between the keywords and provides a ratio in the range $[0, 1]$.

The technique picks under-clustered requests by considering all single item clusters

and the remaining larger clusters as over-clustered requests. For the clustering, we chose[1] a low threshold ($t1$) and a high threshold ($t2$). For the agglomerative clustering, the condition $(<, t1)$ is used for under-clustered requests to cluster nearly similar requests together. Similarly, condition $(>, t2)$ is used for over-clustered requests to break apart requests which are very different. At the end of this step, we obtain clusters where requests corresponding to the same action are clustered together.

4.3.2.3 Action Mapping

The goal of this step is to find a correspondence between the actions from the desktop and mobile web applications. As shown on lines 4—11, this is achieved by using the function $isSimilar$, which checks the similarity of request clusters across the two platforms to establish a mapping. For this purpose, this function applies the Jaccard distance metric to the set of words associated with the requests of each cluster by using the low threshold ($t1$) from the previous step. If one cluster matches to a single cluster from the other platform, a mapping is added between those clusters. In the case, where this mapping is overlapped over multiple clusters on a particular platform, such clusters and any existing mapping is merged. Finally, each unique cluster across both platforms is assigned a unique symbol from the same alphabet. In terms of our motivating example, each requests on lines 1, 7, 9 and 11 will be assigned a unique but same symbol across the two platforms.

4.3.3 Trace Set Canonicalization

The goal of this step is to cluster traces which instantiate the same feature. Then one trace in each cluster can then be retained as the representative of the feature, discarding the others. We will refer to this chosen trace as a *feature instance* or even simply a *feature*, where the distinction is unnecessary. Thus, the output of this step is two sets of feature instances, one corresponding to each platform.

[1]For our evaluation, we empirically picked the values of $(t1, t2)$ as (0.3, 0.8)

This canonicalization is performed by reducing each trace down to the most elemental form of the use-case it represents. To do this our technique finds and removes all repeated action subsequences within each trace. Intuitively, these repeated action subsequences would correspond to repeated portions of a basic use-case, for example, creating multiple blog posts, in the context of our motivating example from Section 3.1.

For finding such repeated sequences, we use an algorithm for finding tandem repeats, which is a popular technique used in biology to find repeated subsequences in a DNA sequences [8]. In general, a *tandem repeat* is a set of two or more contiguous repetitions of a sequence. The algorithm iteratively finds the occurrences of such repeats and replaces them with a single instance of the sequence. After performing this reduction for each trace in our trace sets, any duplicate traces thus created are removed from the trace set, thereby retaining only one feature instance per potential feature.

As an example, consider the sequences `(AQRRS, AQRS)` from the high-level overview presented in Figure 16. The technique will first replace the tandem repeat of subsequence `RR` in the first trace with `R`. The resulting two sequences would then be identical and hence merged into the same feature instance, as shown in the next step in the figure.

4.3.4 Feature Matching

The goal of this step is to find a one-to-one correspondence between the two feature instance sets (from the desktop and mobile versions of the application) created in the previous step. This implies a matching of the corresponding features represented by each feature instance. We formulate this feature instance matching problem as a *maximum weighted bipartite matching (MWBM) problem*, which is a well known problem in the field of operations research. Given a bipartite graph $G = (V, E)$,

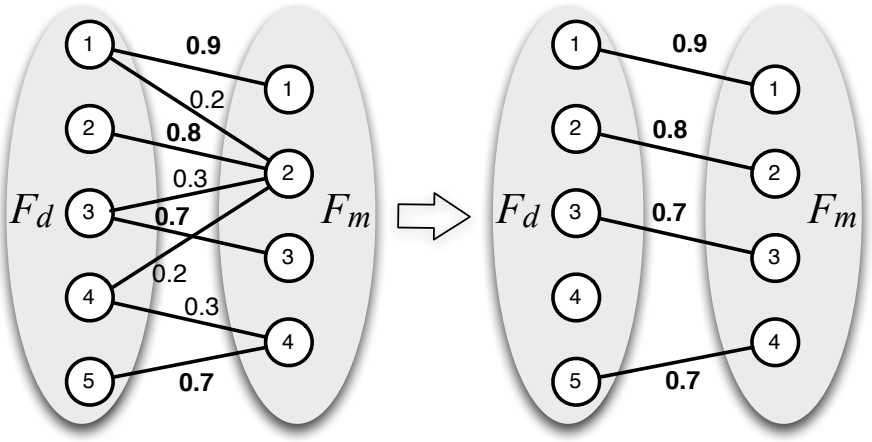


Figure 17: Bipartite graph of features

with bipartition (D, M) and a weight function $w : E \mapsto \mathbb{R}$ the MWBM problem is to find a matching of maximum weight where the weight of matching $\mathcal{M}$ is given by $w(\mathcal{M}) = \sum_{e \in \mathcal{M}} w(e)$. The most popular solution to this problem is presented by the *Hungarian algorithm*, which has been applied to instances of this problem for transportation planning and assignment of agents to tasks [38, 49]. We use this algorithm in our implementation as well.

In our formulation of the MWBM problem, we create the bipartite graph G with one vertex for each feature instance. Thus, the set of vertices D and M, forming the biparition, denote the feature instances from the desktop and mobile versions respectively. The edges E running between D and M denote the possibility of matching the corresponding features and the weight on an edge denotes the profit [2] of matching those two features, in other words the likelihood that they are indeed correct matches.

Figure 17 illustrates this problem formulation. On the left side is an instance of the problem, where features 1-5 from the desktop platform (F_d) are connected to features 1-4 from the mobile platform (F_m) through edges, with profit as labels for each pair. On the right side of the figure is the solution to the MWBM problem where only

[2] A profit function is the inverse of a cost function. Instead of minimizing the cost, the goal here is to maximize the profit.

the edges contributing to the maximum overall profit are retained. This matching is the final outcome of the algorithm and provides a list of matched features, which is $[(1,2,3,5),(1,2,3,4)]$ for the example. The figure also shows unmatched features from both platforms. Feature 4 for desktop is unmatched in this example.

A key step in our formulation is the assignment of weights or "profit values" to the edges of the bipartite graph. This value should reflect the likelihood that two feature instances, each represented by a sequence of actions, are in fact matches of each other. Our solution involves assigning weights to each action in the alphabet and then computing the profit value of a pair of potential matching action sequences as the additive weight of the *heaviest common subsequence* between them. This solution is developed in the following sections.

4.3.4.1 Assigning Weights to Actions

Since we cannot directly identify service invoking actions in a feature instance (*i.e.,* a trace) versus ones that perform navigation or request presentation resources, we cannot use Definition 10 to directly compute feature matchings. However, we exploit the observation that actions specific to exercising specific services would only be observed in use-cases using that service. Thus, rare actions and unique action sequences can be and often are the signature of a feature.

Hence, our technique assigns a weight to each action based on how many times it occurs across different feature instances on that platform. In particular, we use the following formula to compute weight:

$$\omega(a) = 1 - \frac{count(F, a)}{|F|}$$

where, $\omega(a)$ is weight of action denoted by symbol a, $count(F, a)$ is a function that computes the number of feature instances out of all feature instances (F), which contain a, and $|F|$ denotes the total number of feature instances. Thus, if an action occurs in all features its weight will be zero. However if it occurs in fewer features it

will be assigned a weight closer to 1. Once these weights have been assigned, these are used to compute the heaviest common subsequence match from respective traces.

4.3.4.2 Heaviest Common Subsequence

The Heaviest Common Subsequence (HCS) problem aims to find a common subsequence among two sequences, which maximizes the additive weight of the items in the common subsequence [34]. The HCS problem can be defined formally using the following recursive formula:

$$W_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ W_{i-1,j-1} + f_{i,j} & \text{if } i,j > 0 \text{ and } x_i = y_i \\ max(W_{i,j-1}, w_{i-1,j}) & \text{if } i,j > 0 \text{ and } x_i \neq y_i \end{cases}$$

where $W_{i,j}$ is the weight of $hcs(x[1..i], y[1..j])$, i.e., it is the weight of the heaviest common subsequence between the prefix of sequences x, y of lengths i, j. The weight function f, which is used in HCS considers the weights of actions from both platforms and is computed as: $f_{i,j} = \omega(x_i) \times \omega'(y_j)$ where, (x_i, y_j) are actions in the features (x, y) respectively at positions (i, j), and (ω, ω') are the weight functions from the two platforms.

The technique computes and stores the associated HCS weight for all pairs of features across the desktop and mobile platforms and stores it in an $N \times M$ matrix, where (N, M) are the number of features on the desktop and mobile platforms respectively. As explained earlier, this weight corresponds to the likelihood of match between the pairs.

4.4 Evaluation

In order to assess the usefulness and effectiveness of our technique we implemented it in a tool called FMAP and used it for our experimentation. Our evaluation addresses the following research questions:

RQ1: How effective is FMAP in recognizing web application actions across different traces and platforms?

RQ2: How effective is FMAP in matching features between the desktop and mobile versions of real web applications?

In order to establish a baseline technique for evaluating RQ2, we explored existing solutions and found that feature matching is currently done manually by developers. Therefore, we used the following baseline technique, which shares the overall framework of our solution but lacks some of its sophistication. Specifically, it works as follows: First, it uses the URLs in network requests to identify actions across platforms. Next, it combines traces on the same platform with identical action sequences into the same feature instance. Finally, it uses the MWBM problem formulation using the *edit distance* metric as the cost function to establish feature matching across the two platforms. These represent reasonable baseline design choices since URL-based servuce identification is commonly used by web developers to report runtime details of a web application, such as web analytics and traffic monitoring. Similarly, edit distance is a commonly used metric for comparing and matching strings and sequences.

In the following sections, we describe the implementation, the test subjects, the protocol for conducting and evaluating the experiments, and the results of the study itself.

4.4.1 Tool Implementation

Our prototype tool, FMAP consists of two components. The first component performs trace extraction and is implemented as an extension for the Chromium web browser (`http://chromium.org`). It performs the following tasks for implementing trace extraction: 1) To capture the network request-response information, it attaches to the browser's debugger interface, 2) It allows the user to select the desktop or

mobile platform and alters the HTTP user agent string in the network requests to emulate the iPhone 5 mobile browser, 3) Once the trace is captured, it takes the use case name specified by the user and saves the trace to a file in JSON format, and, 4) It also resets the browser state after each trace is captured by the user. In addition to these activities, FMAP also saves a screen dump as the user navigates between different screens of the web application. This screen dump functionality is independent from the technique and is used for verifying the results of our experiments.

The next component of FMAP is written in python and implements the action recognition, feature identification, and feature matching steps of the technique. During the trace simplification step of action recognition, it first extracts the set of keywords from each request. To detect the several inflected forms of words as one, the words are reduced to their root forms by using the WordNet lemmatizer [47] in the Python natural language toolkit (`http://nltk.org`). All metrics were computed by using the corresponding methods from the `nltk.metrics` package. We have in-house implementations of the Clustering and Heaviest Common Substring algorithms, in python. Finally, we used the open source python library, `munkres`, suitably modified to handle floating point profits, for solving the MWBM problem.

4.4.2 Subjects

The ultimate goal of our technique is to match features between desktop and mobile versions of a web application, which may *appear* substantially different, but intrinsically embody similar features and functionality. To perform a meaningful evaluation of our technique, we selected nine web applications whose mobile and desktop versions appear to be quite different. These applications are listed in Table 5. The first six subjects are popular open source web applications obtained from `http://ohloh.net`: `wordpress` version 3.6, a web blogging tool; `drupal` version 7.23, a content management app; `phpbb` version 3.0, a bulletin board; `roundcube` version 0.9.4, an email

client; `elgg` version 1.8.16, a social networking app; and `gallery` version 3.0.9, a photo sharing app. These applications were configured with specific mobile presentation plug-ins and set up to run on a local web server. In particular, we used the wordpress mobile pack version 1.2.5, nokia mobile theme version 6.x-1.3 for drupal, artodia mobile style version 3.4 for phpbb, mobilecube theme version 3.0.0 for roundcube, elgg mobile module version 2.0, and imobile theme version 2.7 for gallery. The three other subjects, `wikipedia.org`, `stackoverflow.com`, and `twitter.com`, are public websites from the Alexa top website list. We chose these sites in particular because they demonstrated significant differences in appearance on the desktop and mobile application versions and were quite different from our existing open source applications.

4.4.3 Protocol

To collect the experimental data for our evaluation, five graduate students were recruited. At first, they installed a fresh version of the Chromium web browser to ensure that the collected data is not corrupted by existing user sessions and extensions. Next, they installed FMAPś browser-extension component. To ensure that our traces do not suffer from biased usage, we asked different students to independently access *all* use-cases of the desktop and mobile versions of the subject applications. In case the same student accessed both versions, we asked them to create separate users and to provide different data on each platform. For expressing the intended use-case for each trace, the students were instructed to provide the use-case name for each trace. For isolating the effects of use-cases on each other locally, students used the provided functionality in the plug-in to clear the entire browser history before capturing each trace.

The collected traces submitted by the students were then provided to FMAP and the baseline tool to compute the feature matchings. To evaluate the effectiveness of

Table 5: FMAP's Details of subjects and action recognition.

Name	*Type*	*#Traces*		*#Requests*		*#Actions*		*Action F-score*		*#Features*	
		D	*M*	*D*	*M*	*D*	*M*	*D*	*M*	*D*	*M*
`wordpress`	Blog	40	12	415	98	72	12	99.7%	100.0%	29	8
`drupal`	Content	16	15	140	62	32	23	100.0%	100.0%	13	13
`phpbb`	Forum	12	12	230	152	20	19	99.6%	99.3%	11	11
`roundcube`	Email	11	13	144	169	20	24	99.8%	100.0%	6	7
`elgg`	Social	13	9	225	121	39	27	100.0%	100.0%	9	7
`gallery`	Media	37	4	390	117	77	14	99.9%	100.0%	31	4
`wikipedia.org`	Content	60	22	709	162	67	40	99.7%	98.8%	11	10
`stackoverflow.com`	Q&A	19	14	174	104	54	37	97.9%	98.9%	18	14
`twitter.com`	Social	19	14	285	54	73	26	83.5%	99.2%	16	11
Total		**227**	**115**	**2712**	**1039**	**454**	**222**	**97.8%**	**99.6%**	144	85

the tool, we manually analyzed the results and compared them against the use-case names provided by the user. We also checked the screen dumps for the matched use-cases when the provided use-case name was not descriptive enough. The results from our analysis are presented in the next section.

4.4.4 Results

To answer RQ1, we ran FMAP on the subject traces and analyzed the intermediate results generated by the action recognition step. In particular, we obtained a list of all action symbols and the clusters of requests corresponding to them, and compared them against manually computed results. To report the quality of clustering we use the F-score metric [42], which considers both intra-cluster similarity and inter-cluster difference. Since, F-score is a weighted average of both precision and recall of clustering, a higher F-score value indicates better clustering. The results for RQ1 are presented in Table 5, which shows, for each subject, its name, its type, the total of number traces captured ($\#Traces$), the number of requests across all traces ($\#Requests$), the number of actions recognized ($\#Actions$), the computed F-score for action recognition (*Action F-score*), and the number of features identified ($\#Features$). Each of these are listed in the table for both, the desktop (D) and mobile (M) platform. As shown, FMAP was able to reduce 2712 requests on the

Table 6: FMAP's Results of feature matching compared to state-of-art.

Name	*Features Matched* (*Baseline*)											*Features Matched* (FMAP)												
	Rep		*TP*		*FP*		*FN*		*TN*		*F-score*	*Rep*		*TP*		*FP*		*FN*		*TN*		*F-score*	*Mis*	*Ack*
	D	*M*	*D*	*M*	*D*	*M*	*D*	*M*	*D*	*M*		*D*	*M*	*D*	*M*	*D*	*M*	*D*	*M*	*D*	*M*			
`wordpress`	8	8	3	3	5	5	2	1	21	1	48.0%	8	8	7	7	1	1	0	0	21	0	93.3%	21	15
`drupal`	12	12	12	12	0	0	0	0	0	0	100.0%	12	12	12	12	0	0	0	0	0	0	100.0%	0	-
`phpbb`	3	3	3	3	0	0	9	9	0	0	40.0%	10	10	10	10	0	0	1	1	0	0	95.2%	0	-
`roundcube`	10	10	4	4	6	6	0	0	0	0	57.1%	4	4	4	4	0	0	2	3	0	0	76.2%	0	-
`elgg`	9	9	2	2	7	7	4	0	0	0	30.8%	5	5	5	5	0	0	1	1	3	1	90.9%	0	-
`gallery`	0	0	-	-	-	-	-	-	-	-	-	3	3	2	2	1	1	1	1	26	0	66.7%	26	20
`wikipedia.org`	17	17	4	4	13	13	1	4	8	1	34.0%	7	7	7	7	0	0	1	1	3	2	93.3%	2	1
`stackoverflow.com`	13	13	3	3	10	10	4	1	1	0	32.4%	10	10	9	9	1	1	1	1	7	3	90.0%	3	1
`twitter.com`	0	0	-	-	-	-		-	-	-	-	2	2	2	2	0	0	8	8	6	1	33.3%	4	3
Total	**72**	**72**	**31**	**31**	**41**	**41**	**20**	**15**	**30**	**2**	**51.5%**	**61**	**61**	**58**	**58**	**3**	**3**	**15**	**16**	**66**	**7**	**86.3%**	**56**	**40**

desktop into 454 actions with an overall F-score of 97.8%. On the mobile, 1039 requests were reduced to 222 actions with overall F-score of 99.6%. These actions were used to discover 144 features on the desktop and 85 features on the mobile versions of the web applications respectively.

For addressing RQ2, Table 6 presents the effectiveness of the FMAP against the baseline. The table shows, for each subject, the features matched by using the baseline and FMAP, in terms of the number of matchings reported (*Rep*), true positives (TP), false positives (FP), false negatives (FN), true negatives (TN), and the overall F-score of the matching result. To contrast the matched features in different platforms, we report these results for both, the desktop (D) and the mobile (M) platform. In addition, for FMAP, we also report the sum of the missing features across both platforms (Mis), which were verified by us manually, and the number of these features (*Ack*), which were also reported by end users, or acknowledged or fixed by developers in a later version. As shown in the results table, FMAP was able to successfully match features across the desktop and mobile platforms for each of the subjects considered. It reported a total of 58 true matchings with a total F-score of 86.3%. In comparison, the baseline produced 31 true matchings with 51.5% F-score. These results are further discussed in the next section.

4.5 Discussion

Based on the results of our empirical study, we observed that the action recognition step was indeed effective in mapping several requests into the same canonical action. For all nine subjects, FMAP clustered similar requests while achieving high F-scores on both desktop and mobile platforms. The few errors in clustering can be attributed to the cases where the requests contained a lot of user supplied data, which resulted in FMAP classifying them as separate actions in the action clustering step. We noticed that, although FMAP removes a significant portion of such information in the trace simplification step, it is limited by its blackbox view of the application. Future improvements to this step can be made by leveraging runtime information from the application. In particular, dynamic tainting [17, 31] can be used to track the sources of such user supplied data and remove them from the requests before clustering them.

In the matching step, FMAP was effective in matching features from all subjects with significantly higher F-score than the baseline. For drupal, the baseline peforms just as good as FMAP. In this case, the request URL paths could uniquely identify the feature, which is an ideal scenario for the baseline but not common practice. By contrast, in case of gallery and twitter, the baseline could not compute any matchings and hence, no results were reported for them.

The true negatives of matching represents features which were not matched by FMAP and are potentially missing. Our analysis of this result revealed several missing features as reported in Table 6, which were also acknowledged by developers or end-users of the application. For our first subject, Wordpress, we found that the users of the mobile toolkit were frustrated with the absence of certain features on the mobile version of the application [4]. Specifically, users complained about not being able to upload media or add categories to posts on their mobile blog [1]. We also found several complaints from the users who wanted to access administration

features on the mobile version [2]. Some users even stated that they would abandon this software due to its missing features on mobile. In the case of gallery, we found that the mobile version only had features for viewing the photo gallery on mobile, while features for uploading the photos and for performing administrative functions were only available on desktop. We validated the need of these missing features on the project's support forum [12] and found that several users complained about not being able to upload photos, share pictures, comment on gallery pictures, and change settings through the mobile version of the site. In the case of Twitter, we confirmed 4 missing features from output generated by FMAP. These features were related to the viewing or editing the profile details of the logged in user. Although, we could not get access to Twitter's private support requests, we found several users complaining about these features on public forums. Interestingly, we later found that Twitter developers implemented 3 out of these 4 features in their latest mobile web application, namely, the ability to see the current user's favourite tweets, the list of followers, and other users being followed by the current user. We believe that this is an affirmation of the usefulness of the missing features identified by FMAP.

The true negatives reported by FMAP also included few features that were indeed present on both platforms. In these cases, we found that our user missed capturing it on one of the platforms. Our investigation of such cases with our study recruits revealed that such misses were mainly attributable to the complex user interface of the application on the platform in question. Hence, the user could not locate the feature during the trace collection. We believe that this itself might be important feedback for the developer of the application to improve the usability of the user-interface.

With the exception of `twitter`, all other subjects have low false negatives. On analyzing the traces from `twitter`, we found that both the mobile and desktop versions were constructed independently even on the server side — a design which deviates from the One Web principle, upon which our technique is predicated. In spite of this

difference, FMAP was able to match two features on each platform with no false positives. We believe that the duplication of the server-side is unlikely in a general setting in practice, where a single code base favors code re-use and maintenance.

Overall, we think that the results are encouraging and provide clear evidence of the effectiveness of FMAP in matching as well as finding missing features.

4.5.1 Threats to Validity

In this section, we describe details of each threat and how they were addressed in defining and evaluating FMAP. As with all studies, the external validity of subjects will increase with more subjects and experimentation. However, we argue that more than the subjects, the success of the technique relates to the similarity of the cross-platform interface, which is in line with the One Web principle. Another valid concern is regarding the use-case coverage achieved by traces collected by our human subjects. As mentioned earlier, our core feature matching technique is independent of the collected traces. Since it operates on these traces, any improved approach to obtain high coverage traces will also benefit our technique. We also performed a sensitivity study by independently varying $t1$ and $t2$ in Algorithm 4 by ± 0.1, and observed that it did not significantly change the clustering result. For addressing internal validity, we removed any selection bias by picking top web applications, which had dynamic features and demonstrated a clear difference in appearance across platforms.

CHAPTER V

TOWARDS TEST SUITE MIGRATION BETWEEN MOBILE PLATFORMS

In this chapter, I present MigraTest, a preliminary technique to assist migration of test cases of a mobile application across different platforms. In particular, given two platform versions of an application, and a test suite for one, the goal of this technique is to automatically generate the test suite for the other platform version. The underlying intuition for this technique is that the test suites check the application for high-level requirements, which are expected to be very similar across the platforms. Thus, by comparing the behavior of the original test suite on the first platform with the behavior of the application on the second platform, the technique can leverage the similarities in the application versions to generate test cases for the second platform. However, establishing this behavioral correspondence between the two platforms is the most challenging in this context, where the applications are developed independently using totally different technologies.

Typically, while developing mobile applications for two different platforms, the developer needs to develop most functionality on each platform from scratch. The developer also needs to migrate existing software artifacts, such as test suites, between these two platforms. Test suites are needed to ensure quality of the application on all platforms. Moreover, existing test suites embed domain knowledge and exercise the behavior of the application in a meaningful way. Hence, it is desirable to reuse test suites and possibly other artifacts across platforms. Manually migrating such test suites is a human intensive task and is error-prone. Hence, the migration activity should involve as little of manual effort as possible and should be able to perform

most tasks automatically.

However, although desirable, test migration between mobile platforms is a very challenging problem for several reasons. The first and foremost problem is that since the two versions are developed completely independently, often using different languages and application frameworks, the structure of the implementation cannot be used as a basis for migration. Secondly, test cases are sequences of actions, and the space of such action sequence combinations is huge and can be potentially infinite. Hence, a technique that needs to explore the application's state-space, needs to work with a partial model of the appication for tractability.

To address these challenges, we present MigraTest a preliminary technique that performs test migration across different platform versions of an application. At a high level, MigraTest operates in the following steps: (1) *Test trace generation:* The available tests for the source platform are executed on the relevant app version to extract a trace of the test, (2) *Guided model generation:* The state-space of the app is explored on the target platform to generate a partial model of the application along with a mapping to its input trace, and (3) *Test generation:* The mapping is utilized to generate executable tests for the target platform.

Essentially, the MigraTest technique follows an iterative approach to explore just as much of the state space as needed and is directed towards maximizing the number of test steps migrated. In situations when the technique has several potential partial solutions, it tries to evaluate which one out of them is the best by prioritizing exploration under the most promising path through the state space. If the technique generates multiple solutions, it presents all such solutions to the user to let them choose the correct solution.

The main contributions of this work are:

- The introduction and definition of the problem of migrating tests between different, platform-specific versions of a mobile application.

- The definition of a technique for performing cross-platform mobile application test migration.
- The development of MigraTest, a prototype implementation of the test migration technique for translating a mobile application's test suite from its iOS version to the Android version.
- A preliminary empirical evaluation of our technique on three real-world mobile applications for iOS and Android platforms.
- A set of open problems and challenges, that need to be addressed by future work for improving test migration across platforms.

5.1 Motivating Example

In this section, I present a simple mobile application on two platforms. I will use this application to demonstrate the challenges for test migration. This example will be referred to as *MyList*, a mobile app to manage a list of items. Users of this application can add items to the list, mark an item with a "star" or delete an item.

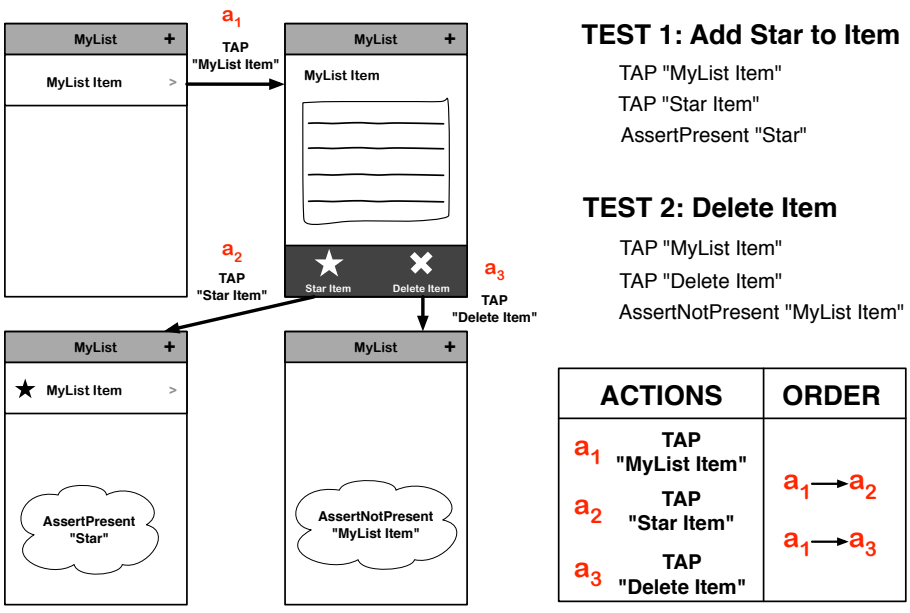


Figure 18: Test cases and partial application state-space for *MyList* on Platform 1.

Figure 18 shows two test cases, and a partial state-space of the mobile application on platform 1. As shown in the figure, the test cases consist of two actions each and have an associated test oracle. The first test performs the action of selecting an item by tapping on the item (action a_1), then it performs another tap on the "Star Item" button (action a_2), and finally checks if the star was actually added to the item. The second test, also selects the item, but then performs a tap on the "Delete Item" button (action a_3), and checks if the item was deleted from the list.

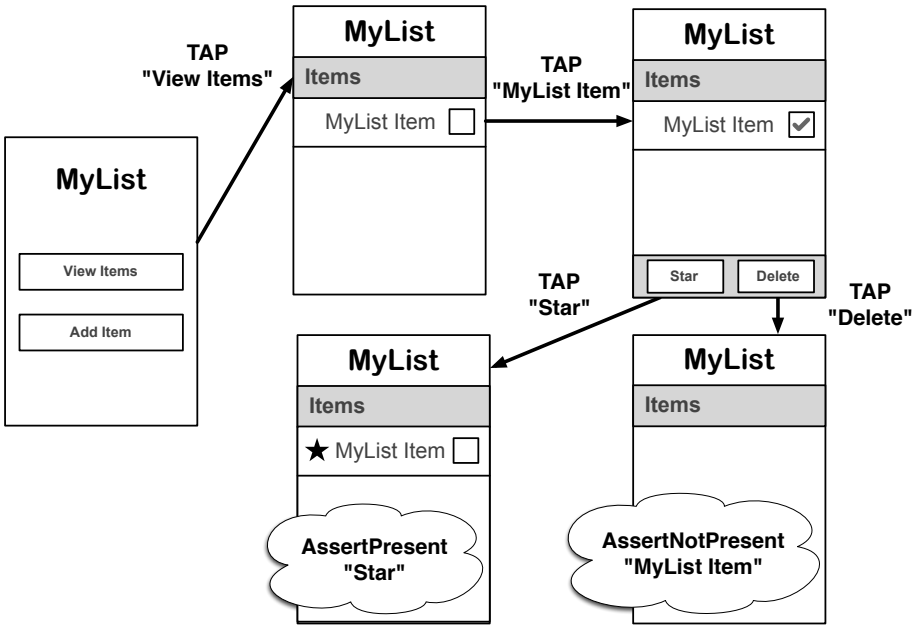


Figure 19: Partial application state-space for *MyList* on Platform 2.

The corresponding application running on platform 2, is shown in Figure 19. The figure shows the partial state-space of the application, which covers the functionality accessed by the two tests on platform 1. Unlike in platform 1, the application on platform 2 has a landing screen on which the "View Items" button needs to be clicked to access the list. After clicking this button, the list item needs to be selected for the "Star" and "Delete" buttons to appear on the screen. Upon clicking these buttons, the selected item is starred or deleted from the list, which is again checked by similar oracles as in the original test cases.

As mentioned earlier, exploring a minimal and partial state-space of the application to cover only the functionality accessed by the tests, is non-trivial without any prior knowledge. As shown in the figure, the function of the "View Items" action is to navigate the user to the item list screen. To capture this partial state-space into a model, the model generator will need to keep track of actions that have been covered, and those which need to be found during the exploration. Moreover, the widgets used to perform the actions across platforms are different (e.g., "MyList Item" is a table cell on platform 1, as compared to a checkbox on platform 2). Additionally, the widgets often have different labels too (e.g., "Star Item" on platform 1, compared to "Star" on platform 2). Such differences make it challenging for the technique to map actions across the platforms, and thus migrate the test cases.

In the next two sections, I will introduce the terminology and the assumptions, which will be then used in the definition of the MigraTest technique for performing test migration.

5.2 Terminology

The terms defined in this section are in the context of User Interface (UI) testing of the application. UI tests interact with the components (or so called widgets) of the application's graphical user interface (GUI) represented on the screen and check the resulting GUI output, just as a user would see it. Hence, UI tests are typically end-to-end tests, which test the behavior of the entire application from a user's standpoint.

Test Action: A test action is an atomic interaction with the application under test, which leads to a change in the application's GUI state. A test action a is defined as the following sequence:

$$a = < actionType, element, actionParams >$$

where *actionType* is the kind of interaction performed, *element* is the information to identify the element upon which the interaction is performed, and *actionParams* is the list of optional parameters, which is sent as a part of the action.

Test Oracle: A test oracle is a special type of action, which checks the assertion to decide success or failure of the test. A test oracle is defined as the following sequence:

$$a = < assertionType, element, expectedValue >$$

where *assertionType* is the kind of assertion that is being performed upon the *element*, and *expectedValue* is the optional value if the assertion type checks for the equality or inequality of the value contained by the *element*.

Test Case: A test case is an ordered sequence of test actions, which corresponds to a use-case of the software under test. The test case actions encode the program inputs, which are supplied to the software to check its behavior against expectation. Formally, a test case (tc) can be defined as a sequence:

$$tc = < a_1, a_2, \ldots, a_n, O >$$

where $a_1, a_2, \ldots, a_n$ are test actions and O is the test oracle and checks the actual behavior against expectation.

Test Suite: A test suite is a set of test cases. A test suite may group test cases based on execution conditions, including environment assumptions or configuration of the software under test. A test suite (ts) is defined as:

$$ts = \{tc_1, tc_2, \ldots, tc_m\}$$

Test Trace: An execution trace from running a test case, or test trace, encodes low level details about the user interface elements (or widgets), which were acted upon

by the test case. A test trace is defined as:

$$tt = < a'_1, a'_2, \ldots, a'_n >$$

where a'_i represents the action along with run time properties of the widgets upon which the action or oracle checking was performed. Specifically, a'_i is of the form $<$ *action_type*, (*widget_type*, *widget_label*, *widget_name*, *widget_value*, ...)$>$.

LTS Model: We model the GUI as a *Labeled Transition System (LTS)*. The LTS is a model whose states are labeled with a set of enabled actions (or transitions). Similar model of computation has been used in prior work for modeling GUI applications for testing [14, 15, 36]. Essentially, an LTS model is a tuple (S, s_0, A, δ), where

- S is a set of GUI states,
- $s_0 \in S$ is the initial state,
- A is the set of actions,
- $\delta : (S \times A) \rightarrow S$ is a state transition function.

Model Trace: A trace in the model is a possible sequence of actions in the model starting at s_0. Formally, a model trace (mt) is defined as:

$$mt = < a_1, a_2, \ldots, a_u >$$

where $\exists a_1 \ldots a_u \in A$ and $\exists s_0, \ldots, s_u \in S \mid \bigwedge_{i \in [1,u]} \delta(s_{i-1}, a_i) = s_i$

5.3 Assumptions

The following assumptions were made during the design of MigraTest. These assumptions are based on our practical experience of how mobile applications are designed for different platforms. These have also been exemplified in the evaluation of the technique, as reported later in Section 5.6.

Action correspondence: Actions, when present on two platforms, have a one-to-one correspondence across both platforms. The goal of the action matching is to establish this correspondence. $Map : A_{p_1} \rightarrow A_{p_2}$, where Map is the matching function and (A_{p_1}, A_{p_2}) are the sets of actions on the two platforms (p_1, p_2).

Although, in theory multiple actions can implement a particular functionality, leading to the same state change, our technique is interested in finding the one-to-one correspondence, which holds across the two platforms and facilitiates maximal test migration. This assumption, not only makes the technique tractable, but also lets it focus only on the best pairs of corresponding actions.

Action types: Actions can either be *key actions* or *navigation actions*. The former actions are crucial to a particular use case in the application, while the later are platform-specific actions to reach a state with an available key action. Our technique relies on the assumption that such key actions need to be present on both platforms, while navigation actions might be absent on either of them. This is guided by our observation of real applications, where the navigation of the application might change across platforms, but the essential functionality is retained through the sequence of *key actions*.

Action ordering: Key actions, which are integral to a test case, occur in the same order on each platform test case and can be separated by a finite number of navigation actions. If tc_{p_1} and tc_{p_2} are two matched test cases for platforms p_1 and p_2, then the ordering of the key actions in each test is preserved. Formally,

$tc_{p_1} = a_1, a_2, \ldots, a_m, O_a$

$tc_{p_2} = b_1, b_2, \ldots, b_n, O_b$

$(\forall a_i, a_j \in tc_{p_1}) \wedge (a_i < a_j) \implies (\exists b_x, b_y \in tc_{p_2}) \wedge (b_x < b_y)$, where $(a_i, a_j), (b_x, b_y)$ are key actions in the application on platforms p_1, p_2 respectively, and $<$ denotes the ordering constraint between actions.

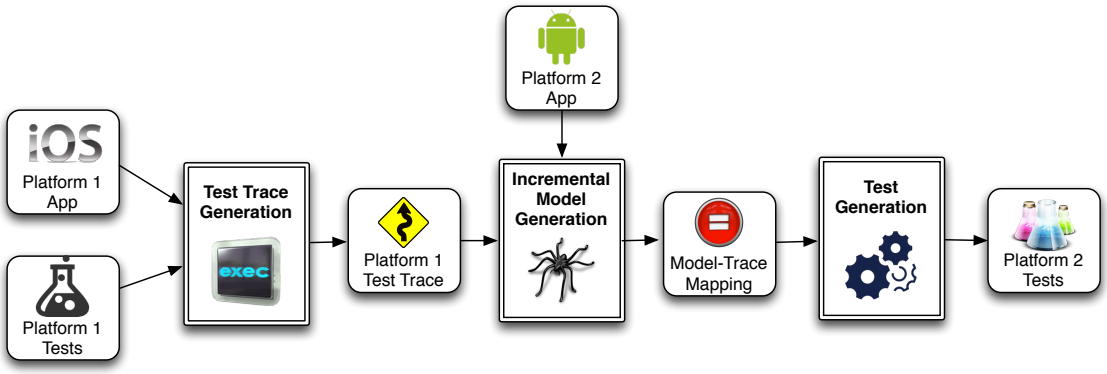


Figure 20: High level overview of the MigraTest approach.

Other than these assumptions, there is also an implicit requirement that the functionality behind the test case is present on both platforms for the test migration to be feasible. This functionality should be reachable from the initial state of the application for the technique to discover it. In addition, the technique harnesses the similarity between action widgets across platforms, to prioritize exploration in the target platform. Thus, it will be more efficient in the presence of such similarities in widgets across platforms.

5.4 Technique

In this section, I present the approach for automatically migrating test cases across two multi-platform versions of a mobile application. As shown in the high level overview in Figure 20, the technique consists of three steps which are described in detail below.

5.4.1 Test Trace Generation

The goal of this step is to capture information from the test cases on platform 1 (i.e., the source platform), which will be used to explore the application state-space on platform 2 (i.e., the target platform). For achieving this goal, the technique transforms the test cases to log information about the widgets upon which the actions are performed at each step. This is a simple source-to-source transformation, which transforms a

statement of the form *widget.action*() to another statement *doAction*(*widget*), where *doAction* is a proxy function defined by MigraTest which logs information about the widget before performing the action. Specifically, *action* can be any UI event supported by the source platform and the proxy needs to understand its semantics to gather information about the widget upon which the action was performed. e.g., Apple's iOS supports widget actions like tap, doubleTap, twoFingerTap and touchAndHold, for which the goal of this step is to log the information about the widget before these actions are performed on it. The information captured about the widget includes two kinds of details: (1) Identifiers, which are used to locate a widget on the UI screen including its accessibility label, name, contained value, or its position within its parent, and (2) Widget state, which is checked by the oracle and includes properties like isValid, isEnabled, isVisible, isFocussed, and, isSelected. All this information is used in the next step to generate the application model on the target platform.

5.4.2 Guided Model Generation

The goal of this step is to explore the partial state-space of the mobile application in the target platform to cover the functionality accessed by the test cases. To achieve this goal, MigraTest interacts with the user interface of the application, as a black-box, and learns an LTS model from the traces explored. During this model generation, MigraTest guides the exploration of the application's user interface by considering the current mapping of actions from the input test traces to the model and the dependent, unmapped actions in the test traces. The technique is described in the overall algorithm below.

5.4.2.1 Overall Algorithm

Consider an application on two platforms p_1 and p_2, with test traces from p_1 in the form $tt =< a_1,\ a_2,\ \ldots,\ a_m >$. The goal of the technique is to generate an LTS model

from p_2 in the form (S, s_0, A, δ), with a maximal mapping between the actions in the trace and those in the model (i.e., A). To achieve this goal, the technique explores the application state-space by interacting with the application, and at each step chooses which action to perform, for covering the required functionality.

Algorithm 5 shows the overall algorithm behind the MigraTest technique. As shown, the technique takes as input, the test traces (TT) from running the tests on the source platform. The model is initialized with the default state of the application and all the active widgets, upon which actions can be performed, are added to a $Worklist$ (lines 2-3). Then the technique iteratively selects each widget and performs the suitable actions on it to reach new states (lines 5-7). At this step, the technique first tries to maximize the mapping between trace actions and actions in the model in $MaxMatchSolve$. Details of this matching are described later in 5.4.2.2. Once this analysis is complete, the next widget action is selected by the $getNextAction()$ function, which analyzes available actions and their dependence on matched actions. This selection is done by assigning a priority to the available widgets, as described in 5.4.2.4. After the selected action is performed, the resulting state is compared to the set of known states, and if it is a newly discovered state, then its widgets are added to the $Worklist$ (lines 8-9). At each step, the information is added to the model and the model exploration is repeated until all actions have been performed or if the $TIMEOUT$ is reached. Finally, the technique returns the captured $Model$ and computed mapping Map between the trace and model actions on line 11.

5.4.2.2 Action Mapping as an Optimization Problem

This subsection describes the logic behind the maximal matching function $MaxMatchSolve$. This maximal mapping between actions of the two platforms should be such that each test trace on p_1 can be mapped to a model trace in p_2. The problem is formulated as the following optimization problem:

Algorithm 5: Guided Model Generation

Input : TT: $\{< a_1, a_2, \ldots, a_n >, \ \ldots\}$ // Set of test traces from the source platform
Output: $Model$: (S, s_0, A, δ) // Explored application model for the target platform
Map: Action mapping between source and target platforms

```
1  begin
       // Initialize Model
2      Model ← (S, s_0, A, δ)|S = {s_0}, A = ε, δ = ε
3      Worklist ← s_0.actions()
4      while Worklist.size > 0 and time() < TIMEOUT do
5          Sol ← MaxMatchSolve(Model, TT)
6          (s, a) ← Sol.getNextAction(Worklist) // Get Next Action to Perform
7          s_next ← doAction(s, a)
8          if s_next ∉ S then   // s_next is a new state.  Add it to worklist
9              Worklist ← s_next.actions()
10         Model.add(s, a, s_next)
11     return Model, Sol.Map
```

$$\max \sum_{a \in \Sigma tt_i} |Map(a)|$$

such that

- $Map : A_{p_1} \rightarrow B_{p_2}$ is a mapping from every action $a \in A_{p_1}$ from platform p_1 to action $b \in A_{p_2}$ in platform p_2 and $A_{p_2} = \{\varepsilon \cup A\}$, where A is set of actions in the model.

- $|Map(a)|$ is 0 if a is mapped to ε and 1 otherwise

- $(Map(a_i) = b_x) \wedge (Map(a_j) = b_y) \wedge (a_i < a_j) \wedge (b_x \neq \varepsilon) \wedge (b_y \neq \varepsilon) \implies (b_x < b_y) \wedge (\{\forall b_z \mid (b_x < b_z < b_y)\} \wedge \{\exists a_k \mid (Map(a_k) = b_z)\}) \rightarrow (a_i < a_k < a_j)$

The $<$ operator in the last constraint specifies the order between actions, such that $a_p < a_q$ denotes that action a_p is followed by action a_q. The last constraint in the optimization problem enforces ordering on the set of matches produced. In particular, it restricts that if actions a_i, a_j from platform p_1 matches with actions b_x, b_y from platform p_2, then their ordering must be preserved. Additionally, any non ε action a_k on platform p_1, between the matched actions (a_i, a_j), should be either mapped to ε or an action b_z, which is in between actions (b_x, b_y) on platform p_2. This not only ensures that the matched actions appear in the same order in both the test trace

and the model trace, but also that two pairs traces with distinct key actions have non-overlapping distinct model traces.

5.4.2.3 Branch-and-Bound

Branch-and-bound is a general scheme for building algorithms to solve hard optimization problems [18, 39]. Branch and bound follows a divide and conquer strategy to search through the entire space of solutions. Unlike exhaustive enumeration, it uses a bounding constraint to discard a subset of the solution space that does not satisfy this constraint. At each step, the algorithm maintains the status of the solution along with respect to the search in the space of all solutions. This is described in terms of the current best solution and a subset of the unexplored solution space.

MigraTest uses branch-and-bound to compute the matching between all pairs of actions between the two platforms. The bounding condition is governed by the constraints presented in the previous section, which decide feasibility of a possible matching based on the ordering constraints. At each iteration of the guided model generator, MigraTest reports the final branch-and-bound tree, which contains all the feasible matching solutions.

The worst case complexity of branch-and-bound algorithms is typically the same as that of exhaustive enumeration. However, in practice, the bounding constraint limits exploration of all solutions under a particular infeasible partial assignment. Moreover, the branch-and-bound algorithm does not restrict the search strategy in the solution space and there is further room for optimization there to find an optimal solution quicker as described in the next section.

5.4.2.4 Prioritizing Actions during Model Generation

To reach at an optimal solution faster, the technique prioritizes performing actions, which are similar across platforms and can lead to discovery of more actions for better matching. To this effect, the priority of a potential action, b on the target platform

is computed using the following formula:

$$Priority(b) = \sum_{\forall a | Map(a)=b} \left(Similarity(a,b) \times \sum_{\forall a_i | a_i > a \wedge Map(a_i)=\varepsilon} 1 \right)$$

where, the second part of the multiplier in the equation is the count of all actions in the trace, which are dependent on a. This allows the technique to prioritize actions, which would enable more solutions for the dependent actions by discovering more candidate actions in the model.

The first part, *Similarity*, is a function that computes the similarity between the labels of the two cross-platform actions, a and b. It also considers a conservative over-approximation of a set of compatible actions (i.e., a *tap* can be mapped to a *tap*, *longTap* or *doubleTap* but not to a *type* action across platforms). This function is defined as:

$$Similarity(a,b) = \begin{cases} \left(1 - \dfrac{LevensteinDistance(a.label, b.label)}{max(a.label.length, b.label.length)}\right), & \text{if } a \sim b \\ 0, & \text{otherwise} \end{cases}$$

where, if a and b are compatible ($\sim$), then the similarity score is computed based on the Levenshtein distance [40] between the labels. If the actions are incompatible, their similarity score is zero.

5.4.3 Test Generation

The goal of this step is to generate the test cases for platform p_2 from the traces for platform p_1 and the solution generated by the guided model generation step. Specifically, the technique maps each test trace to a model trace by leveraging the mapping, and replaces the test actions to the corresponding model actions. In addition to mapped actions, the technique adds any extra actions in the model trace on platform p_2, which are needed to connect mapped actions, or the initial state to the first action

in the model. Actions from platform p_1, which are mapped to ε are omitted in the process.

While doing performing test generation, the technique maintains any contextual information, which is relevant for the action to be performed on platform p_2. This information includes input values for test actions, which interact with textual elements on screen, and expected values for the test oracles, which check this value against the actual value while performing the assertion. All this contextual information about actions in the model trace on platform p_2, is encoded into executable test cases through code generation. These executables tests are provided to the developer as a final result of the test migration technique.

5.5 Illustration of the Guided Model Generation

In this section, I will explain the implementation of the guided model generation part of MigraTest and illustrate it with the motivating example. As mentioned in the technique section, the goal of the model generation is to expore the state-space of the application and finally generate a model of the state-space along with a mapping of test trace actions to the model.

The input test cases to MigraTest for the *MyList* application on platform p_1, were presented in Figure 18. From this input, it generates the test traces:

$$TT = \{< a_1, a_2 >, < a_1, a_3 >\}$$

where, $a_1 =<$ TAP, (TableCell, "MyList Item") $>$, $a_2 =<$ TAP, (Button, "Star Item") $>$, and $a_3 =<$ TAP, (Button, "Delete Item") $>$ are the three trace actions from platform p_1.

When MigraTest starts its exploration, it sees the initial screen of the mobile application. As an example, refer to the partial state-space for the *MyList* application shown in Figure 21. The model generator sees two actions *TAP "View Items"* (b_1) and

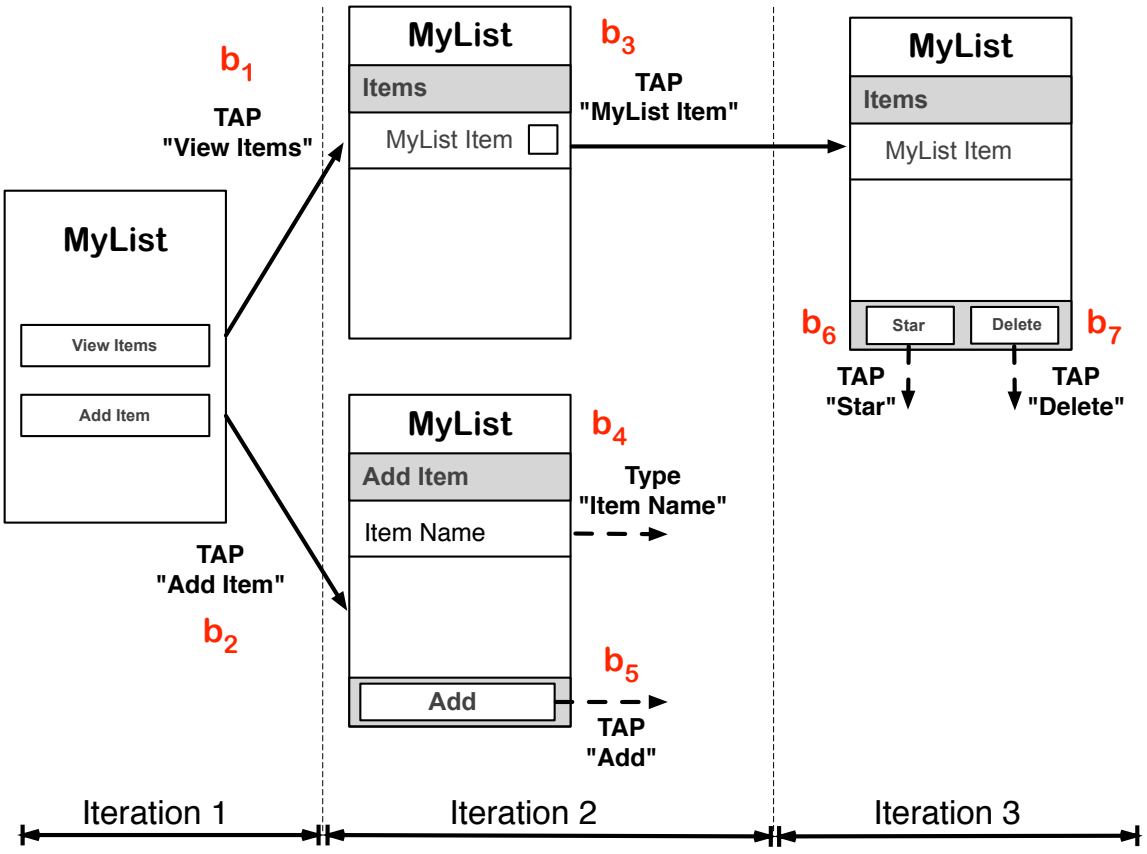


Figure 21: Partial state-space of MyList during model generation.

TAP "Add Item" (b_2) on this first screen. The exploration of the application's state-space is shown over three iterations in the figure, where each new iteration results in a bigger model. During its exploration, the technique keeps track of each pair of candidate actions to be matched, along with their similarity score. This information is maintained in a tabular representation, as shown in Figure 22, for the three iterations of the technique. As shown in the figure, the table contains platform p_1 actions as rows and platform p_2 actions as columns. Thus, as new actions are discovered during the model generation, they are added as new columns to the table.

In the first iteration, two actions b_1 ad b_2 were found but not yet explored by the technique. The similarity scores are listed in the table under this iteration. Under this current state, any action from platform p_1 could be assigned to any other action from platform p_2 individually. However, since there are two orderings on platform p_1,

Figure 22: Tabular representation for matching actions across platforms. Rows represent actions from platform p_1 and columns represent actions from platform p_2.

i.e., $(a_1 < a_2)$ and $(a_1 < a_3)$, either only a_1 or both a_2, a_3 can be matched with either b_1 or b_2. Since, no solution was found for all actions, the technique decides to explore further and needs to decide whether to perform action b_1 or b_2. Computing the Priority gives the same priorities to both these actions, as shown below, and hence both the actions are performed in this iteration.

$$Priority(b_1) = 0.4 \times 2 + 0.5 \times 0 + 0.5 \times 0 = 0.8$$

$$Priority(b_2) = 0.4 \times 2 + 0.5 \times 0 + 0.5 \times 0 = 0.8$$

In the second iteration, after these actions are performed, the technique discovers actions *TAP "MyList Item"* (b_3), *TYPE "Item Name"* (b_4) and *TAP "Add"* (b_5). Since, b_4 is not compatible with any other action on platform p_1, it is assigned a zero similarity score. In addition, the similarity score of b_5 with every action on platform p_1, computes to zero, as well. In the case of b_3, there is a perfect match of action labels with a_1 and thus, its similarity score is 1 under the corresponding place and is fractional with a_2 and a_3. This iteration brings more solutions to the action matching problem. However, the best match for a_1 is with b_3, which still leaves two pending actions that are dependent on it. Note that, there are other possible solutions in this iteration, which make assignment feasible (e.g., $a_1 = b_2,\ a_2 = b_4,\ a_3 = b_5$). however, these assignments are suboptimal to the single assignment of a_1 with b_3. Hence, the technique decides to continue the exploration. The computed priorities for b_3 is 2,

whereas it is 0 for b_4 and b_5. Hence, the technique explores action b_3.

$$Priority(b_3) = 1 \times 2 + 0.4 \times 0 + 0.4 \times 0 = 2$$

$$Priority(b_4) = 0$$

$$Priority(b_5) = 0$$

In the third iteration, after action b_3 is performed, the technique discovers actions b_6 and b_7, each of which has strong fractional similarity scores with a_2 and a_3 respectively. At this point, the technique has found a feasible and optimal matching ($a_1 = b_3$, $a_2 = b_6$, $a_3 = b_7$) in this iteration.

5.6 Evaluation

To evaluate the test case migration technique, we implemented it in a tool called MigraTest and used the to answer the following research questions:

RQ1: Can MigraTest effectively translate test cases, from one version of a mobile application, to semantically equivalent test cases on the other platform?

RQ2: During the test migration, was MigraTest able to reveal any platform specific issues between the two versions of the application?

RQ3: How efficient is the guided model generation phase of MigraTest?

RQ1 addresses the effectiveness of MigraTest in the test migration task with respect to the quality of tests generated and their equivalence to the original tests in the source platform. RQ2 addresses the fault finding ability of MigraTest while performing test migration. RQ3 addresses the efficiency with respect to the time taken by MigraTest to explore the state-space of the application on the target platform and if it minimized the exploration of actions, which were not necessary for the test case migration.

In the rest of this section, we will present details of our implementation, the subject cross-platform apps used for our evaluation, experimental protocol and the results of our study.

5.6.1 Tool

Our prototype tool, MigraTest is engineered to migrate GUI tests from an application written for Apple's iOS platform to a corresponding application written for the Android platform. The tool accepts tests written using the *UIAutomation JavaScript API*, which is a testing library provided by Apple, as a part of the iOS SDK. MigraTest defines proxy functions in JavaScript to capture information for each widget, while performing the action as described in the technique. Each API function that corresponds to a test action is replaced by a relevant proxy function in the iOS test cases. The trace generator is implemented as a shell script for executing the test cases and a python script for parsing the low level traces for extracting relevant information, which is logged by the JavaScript proxy functions. The guided model generator for Android is built on top of the PUMA tool [32]. Specifically, we enhanced PUMA with our optimization engine, which implements the solver and decides the exploration strategy as described in the technique. The test generator supports two test oracle types: (1) *AssertExists*, which checks for the existence of an element on the final screen, and (2) *AssertText*, which checks the textual content of an element. The resulting test cases are generated using the python wrapper for the *uiautomator* tool for Android.

5.6.2 Subjects

The goal of the evaluation is to migrate test cases from an iOS mobile application, to its version on the Android platform. Thus, we selected three mobile applications with both iOS and Android versions. The first application, *wikiHow Survival Kit*, includes

100 articles for everyday emergencies and extreme scenarios. This application has pre-populated data for the users to browse through the information even when offline. The second application, *WhiteHouse*, is the official application for the American president and contains news articles, press briefings and audio-visual materials from the White House. The third application, *BarCodeFitness*, is a fitness application, which helps its user keep track of weightlifting workouts at the Campus Recreation Center at Georgia Tech. In the case of *wikiHow*, the iOS and Android versions of the applications were built by different developers. The *WhiteHouse* app was built by a team of professional developers and the source code was released to the open source community. The iOS and Android versions of the *BarCodeFitness* app were built by different teams of students and we obtained the source code of these versions from our school's App Lab[1].

5.6.3 Experimental Protocol

For our experiments, the subject applications were installed on an Apple iPhone 5S and a Nexus 4 emulator. We recruited 2 graduate students for each application, and provided them with the iPhone to interact with the iOS version of the application. The students were instructed to study the user interface of the application, define as many test cases as they could envision, and submit them to us on a form. We encoded these test cases into the UIAutomation tests in JavaScript. These automation tests were executed, and the resulting trace was captured for the technique. The results reported by the technique were manually analyzed against the input test cases to determine the effectiveness of the technique. For computing the efficiency, we compared the state-space of the applications, which was explored by MigraTest tool in terms of the actions performed on the application, and compared it to the number of actions, that would be needed by the vanilla PUMA crawler to make the test migration

[1]Gatech App Lab - http://gtjourney.gatech.edu/app-lab

possible. We explain the results in the next section.

5.6.4 Results

Table 7: Test cases for the iOS versions of the subject applications.

Subject	Test	Description
wikiHow	Items loaded	When user selects category, check if items in category are loaded
	Page title	When user selects category, check if page title matches category name
	Back button	When user selects category & item, and presses button with label "Back", check if page title matches category name
	Pop-up shown	When user selects category & item, and presses button with label "Add", check if pop-up is displayed to user
	WebView loaded	When user selects category & item, check if a WebView is added on the screen
WhiteHouse	Menu loaded	When the user taps on Menu, check if "BLOG" is present
	Photo loaded	When the user taps on Menu, and selects "PHOTOS" check if the first photo is loaded
	Video loaded	When the user taps on Menu, and selects "VIDEOS" check if the first photo is loaded
	Search	When the user taps on Menu, and enters "A" in the search bar, check if page contains "No Results"
BarcodeFitness	Workouts button	When the user loads the app, the "Add Workout" button is shown
	Exercise button	When the user selects an item, the "Add Exercise" button is shown
	Exercises loaded	When the user selects an item, and presses "Add Exercise", and presses "List", check if workout items are loaded
	Add Workout	When the user selects "Add Workout", and types "Workout1", and presses "Save", check if "Workout1" is added to list
	Add Exercise	When the user selects an item, and presses "Add Exercise", and presses "List", and presses "Ab crunch", and presses "Done", check if "Ab crunch" is added to list

The student participants provided a total of 10 natural language test cases per application, which were used to build the automated test cases. We ignored the duplicate scenarios of tests between the 2 students that analyzed each application. In addition, there were certain natural language tests, which could not be represented by the testing infrastructure on iOS. An example of such an unsupported feature is for performing assertions on the element inside a WebView component in the wikiHow application. This is because, WebView is a browser component used in the application, which is seen as a black box by the test case. Table 7 shows the test cases for the iOS versions of the application, which were considered for our evaluation.

The results of MigraTest for RQ1 are shown in Table 8. The table shows for each test belonging to the three subjects: 1) the name of the iOS test for the subject, 2) if the test was successfully migrated to the Android version of the application, 3) if not migrated successfully, the challenge which prevented the migration, and 4)

Table 8: Test Migration Results

Subject	Test	Migrated	Challenge	Potential Issues Found
wikiHow	Items loaded	Yes		
	Page title	Yes		Duplicates found
	Back button	Partial	Platform specific functionality	
	Pop-up shown	Partial	Missing functionality	Missing functionality
	WebView loaded	Yes		
WhiteHouse	Menu Loaded	Yes		
	Photo Loaded	Yes		Empty label on Android
	Video Loaded	Yes		Empty label on Android
	Search	Yes		
BarcodeFitness	Workouts button	Yes		
	Exercise button	Yes		
	Exercises loaded	Yes		
	Add Workout	Partial	Missing functionality	Missing functionality
	Add Exercise	Partial	Complex Widgets	

any potential issues found in the application, during the migration. As shown in the results, MigraTest was able to migrate 10 out of 14 tests completely. In the cases where MigraTest was able to migrate the test successfully, the migrated test was the most optimal one. In the 4 cases, where MigraTest was unsuccessful in completely migrating the test, we found that it could migrate most of the test actions and faced some challenges in performing further migration, as listed in the table. In the case of *wikiHow*, one test case needed the user to interact with a UI widget with a label "Back". However, on the Android version, such a widget was not present and this functionality was implemented using the hardware `back` button. In the case of two tests, i.e., the "Pop-up shown" test for *wikiHow* and the "Add Workout" test for *BarcodeFitness*, the entire functionality behind these tests were missing on the Android version of the application. Further, the "Add Exercise" test case required the crawler to interact with a complex widget on Android, which was used to select specific values for selecting the details of the exercise. This widget was composed of two buttons, *high* and *low*, and a read-only textual element, whose values could be altered by interacting with the buttons. Without the knowledge of such a composition, the model generator could not intelligently input the value needed by the test case.

In terms of RQ2, MigraTest was able to detect some issues in the test migration process. For the *wikiHow* application, MigraTest reported the presence of a

duplicate entry for the textual value of the element under assertion. Upon further investigation, we realized that this was a result of a data issue, where there was an additional extra erroneous item in the database, for the Android version of the application. Two test cases that MigraTest was not able to migrate were as a result of missing functionality. In such cases, although the widgets were present but the developer had not implemented the underlying feature. Finally, MigraTest reported that on two test cases, the label of the widget under action had an empty label. Such labels are used by accessibility services to help visually impaired users to understand the purpose of such widgets. Without a label, a visually impared user user would not be able understand the functionality of a particular widget or distinguish between multiple widgets.

To answer RQ3, I would like to compare the efficiency improvements over the baseline crawler, PUMA. However, since the vanilla version of PUMA was not able to crawl any of the subject applications, I made several fixes to it to enable it to explore the application. This modified version of PUMA is called PUMA+. In our experimentation, PUMA+ took more than 3x time than MigraTest to cover all the actions needed for test migration. This is because PUMA+ targets all the available actions instead of the actions that are in the original tests.

From these results, I believe that MigraTest is a promising first attempt at the test migration problem for mobile platforms. I believe that MigraTest was mostly successful in the context of mobile applications because of multiple reasons: 1) Although the cross-platform versions of mobile applications are significantly different, they often implement the same use-cases and a high level equivalence can be established between the versions, and, 2) Mobile applications have a simpler user interface, with lesser actions to be performed than other traditional applications. I believe that these two opportunities make MigraTest tractable for mobile application and help it migrate a significant number of test cases for the subject applications.

However, as reported in our results, MigraTest was not able to migrate some of the tests. Most of these failures were due to the limitations of our current implementation and can be improved with suitable engineering. In the next section, I will list these challenges broadly in the context of the test migration and discuss how they can be addressed in future work.

5.7 Challenges

MigraTest is the first technique and tool to migrate tests between mobile platforms. The case study shows promise in the approach. However, there are several challenges that future work should address to make MigraTest applicable in many other settings.

Platform-specific functionality: The "back" button is an example of a platform-specific functionality, which is present in Android and some other platforms but is missing in specific platforms like iOS. On iOS, the back functionality is implemented using any widget available to the programmer. Understanding the functionality behind the button and translating it across platforms will require further analysis of the application. Similarly, test cases that use other platform-specific features, like the features provided by hardware sensors or actuators, will require special handling in order to migrate them across platforms.

Complex Interactions: Currently our implementation can handle simple widgets that accept `tap` or `type` actions. However, in the case of the *BarcodeFitness* app, there was a composite widget that the target test was supposed to interact with. However, without any prior specification of a complex widget, it is challenging for any black-box model generator to identify such widgets and perform these interactions.

Oracle migration: MigraTest relies on a simple oracle that checks the value or state of a particular UI element on a single screen. However, test oracles can be general and more complicated. For instance, an oracle can be defined over multiple screens, which checks for the absence of an element, which was present on a previous screen. In such cases, inter-screen analysis needs to be performed for migrating such oracles. Another example is where the oracle is a function or a complex program, which returns a true/false result. Migrating such oracles would require a use of program analysis and reasoning.

Sand-boxing: In the case of *wikiHow*, the application contained all the data, without communicating with external servers. Other applications may fetch external resources leading to a source of non-determinism. This can adversely affect the effectiveness of the technique if not handled appropriately. Testing is typically performed in a hermetic environment such as a sand-box, which would need to be setup for both versions of the applications. This would be possible by setting up mocks or proxies, which would present the same sand-boxed environment in both platforms.

CHAPTER VI

RELATED WORK

In this chapter, I will discuss the related work from all the three problems targeted by my research.

6.1 XBI Detection

Previous work on cross-browser compatibility testing can be divided into the following four generations of techniques.

6.1.1 Generation 0: Developer Tool Support

A common resource used by web developers are browser-compatibility tables maintained by reference websites such as `http://quirksmode.org` and `http://caniuse.com`. Using these tables, a developer can manually lookup features used by their applications to know if the feature is supported by a certain set of browsers. However, these tables are generated manually and only have feature information for a limited set of browsers. Some web development tools such as Adobe Dreamweaver (`http://www.adobe.com/products/dreamweaver.html`) provide basic static analysis-based early detection of XBIs during development. However, these tools only check for certain types of XBIs related to specific DOM API calls or CSS features not supported by certain browsers. Modern XBIs often arise through a combination of browser, DOM, and JavaScript features, which is outside the scope of such tools.

6.1.2 Generation I: Tests on a Single Browser

Recently, researchers have targeted web application issues, which are specific to single browsers. The technique proposed by Eaton and Memon [24] is among the earliest

works in this area. Their technique tries to identify potentially problematic HTML tags in a given web page, based on a manual classification of good and faulty pages previously generated by the user. However, XBIs in modern web applications are usually not attributable simply to specific HTML tags, but rather to complex interactions of HTML structure, CSS styles and dynamic DOM changes through client-side code. More recently, Tamm [61] presented a tool to find layout faults in a single page, with respect to a given web browser, by using DOM and visual information. The tool requires the user to manually alter the web page—hide and show elements—while taking screen-shots. This technique is not only too manually intensive to scale to large web applications, but also virtually impossible to apply to dynamically generated pages (i.e., most of the pages in real-world applications). Finally, its focus is not specifically cross-browser testing.

This problem has also been targeted from the point of fixing the browser. Specifically, the *Web Standards Project* introduced *Acid Tests* [6], which are a set of 100 tests that check a given web browser for enforcement of various W3C and ECMA standards. Similarly, the *test262* suite [62] (formerly called *SputnikTests*) can check a JavaScript engine (of a web browser) against the ECMA-262 specification. It is noteworthy that in an experiment we ran, Mozilla Firefox 7.0.1 failed 187 of the 11016 tests in the *test262* suite, Google Chrome 15.0 failed 416 tests, and Internet Explorer 9 failed 322 tests. In other words, the JavaScript engines of these popular browsers are not standard and differ from one another. These suites reveal some of these differences and justify the development of techniques to identify XBIs.

All of the above techniques either test a web application within a single web browser or test a single web browser itself. Thus, while such testing may happen to remove causes of XBIs, they are not, strictly speaking, cross-browser techniques.

6.1.3 Generation II: Multi-Platform Behavior & Test Emulation

This class of tools allows developers to emulate the behavior of their web application under different client platforms. Tools such as BrowserShots (`http://browsershots.org`) and Microsoft Expression Web SuperPreview (`http://microsoft.com`) provide previews of single pages, while tools such as CrossBrowserTesting.com and BrowserStack.com let the user browse the complete web application in different emulated environments. A common drawback of all these tools is that they focus on the relatively easy part of visualizing browser behavior under different environments, while leaving to the user the difficult, manually intensive task of checking consistency. Further, since they do not automatically explore the dynamic state-space of the web application, they potentially leave many errors undetected.

6.1.4 Generation III: Crawl and Compare Approaches

This class of techniques represents the most recent and most automated solutions for cross-browser testing. These techniques generally work in two steps. First, the *behavior capture* step automatically crawls and captures the behavior of the web application in two or more browsers; such captured behavior may include screen images and/or layout data of individual web pages, as well as models of user-actions and inter-page navigation. Then, the *behavior comparison* step automatically compares the captured behavior to identify XBIs.

WebDiff [53] uses a combination of DOM and visual comparison, based on computer-vision techniques, to detect XBIs on individual web pages. CrossT [45], conversely, uses automatic crawling and navigation comparison to focus on differences in the dynamic behavior caused by, for example, unresponsive widgets in a certain browser. CrossCheck [51] combines these two approaches and tries to achieve better precision through machine-learning based error detection. However, it still suffers from a high number of false positives and duplicate error reports. WebMate [19], a

recent addition to this class of techniques, focuses mainly on improving the coverage and automation of the automated crawling, and its XBI detection features are still under development. QualityBots (`http://code.google.com/p/qualitybots/`) is an open source project that checks the appearance of a web application in different versions of the Google Chrome browser. The technique uses pixel-level image comparison and is not yet available for use with other families of web browsers. Browsera (`http://www.browsera.com/`), MogoTest (`http://mogotest.com/`), and Browserbite (`http://app.browserbite.com/`) are the very first industrial offerings in this category. They use a combination of limited automated crawling and layout comparison, albeit based on a set of hard-coded heuristics. In our (limited) experience with these tools, we found that these heuristics are not effective for arbitrary web sites. However, an objective evaluation of the quality of these tools cannot be made at this time, since a description of their underlying technology is not available.

6.2 Feature Mapping

Matching different elements of software has been a problem addressed by works on inferring API mappings and in comparing reverse engineered application models, as described below:

6.2.1 Inferring API migration mappings

There is a body of research [27, 50, 68] on inferring mappings between two versions of an API or between two independent implementations of an API, for example in two different languages. Although this problem seems similar to ours, the granularity is completely different. While API mappings are between individual functions (which can be viewed as atomic actions) constituting the API, feature mapping is about mapping use-cases or traces which are sequences of actions. Further, the basis of extracting similarity is also different. API mapping tools such as Rosetta [27] assume that they are given a population of pairs of equivalent traces, one each from the two

API versions. However, such a trace-level correspondence is actually the output of our technique. Our technique is predicated on the assumption that the two versions of the web application may have different client implementations but exhibit substantially similar behavior at the client-server interface. No such interface exists or can be exploited by API matching techniques.

6.2.2 Reverse engineering of web applications

This body of work attempts to reverse engineer a model of a web application, that can then be used as a basis for constructing test-cases for the application. Some representative techniques in this category include the Crawljax tool [46] and the more recent ProCrawl crawler [56] which dynamically crawl a web application to extract a finite state model of its user interface. The WARE approach by Di Luca et al. [21] uses a combination of white-box static analysis and black-box dynamic analysis to extract a UML model of the web application. A somewhat different approach proposed by Elbaum et al. [57] is to use web application user session data to directly contruct test cases. Our work is orthogonal to this body of work in that it starts with a set of use-cases of the web application on each platform, independent of the source of those use-cases. They could be derived from the models constructed by [21, 46, 56], created from session data as in [21], or derived from some other source of manually or automatically generated test-cases.

6.3 Test Migration

Test case repair for regression testing is a widely explored problem in the context of Java programs, graphical user interfaces, and web applications [7, 20, 22, 29, 44, 54]. These techniques try to repair the test cases by either modifying the test oracle or by altering the test steps. Such changes required for test case repair are typically small and are often obtained by analyzing the results of the failed test cases or the

changes introduced in the new application version. However, in the case of test migration, the different platform mobile applications are essentially completely different implementations. Any technique for test case repair is limited in handling such large differences.

Another related area of work is that of automated test case generation. Such techniques either make use of software specifications [63], or program analysis based methods [16] to generate test cases. In an ideal setting, one could possibly generate tests for each of the platforms using these techniques independently on each program version. However, there are several challenges which restrict the applicability of these techniques for multi-platform software. Software specifications are usually not readily available, especially for multiple platforms. Developing such a specification requires additional effort from the developer on each platform. The program analysis based techniques might generate different tests for each of the platform without any correspondence between the tests. In addition, the program analysis techniques are closely tied to the programming language, they are needed to be developed for each platform separately.

CHAPTER VII

CONCLUSION AND FUTURE WORK

This section concludes my dissertation with a summary of my goals and a discussion of its merit.

7.1 Summary

With the emergence of new computing platforms, software applications are increasingly being developed to target multiple platforms. Hence, developers of such software need to duplicate testing and maintenance activities to support the software on different platforms. Often developers are unable to cope with this ever increasing demand and might inadvertently release broken software for certain platforms, or miss deadlines while attempting to address this issue. Hence, in my research, I am developing automated techniques to assist developers with cross-platform testing and maintenance tasks.

The thesis of my research is that approximate behavior matching can be used to develop techniques to address cross-platform testing and maintenance problems. To investigate this thesis, I have developed three techniques for (1) finding inconsistencies in an application that runs across platforms, (2) to detect missing features across two different versions of a multi-platform application, and, (3) to migrate test cases across two platforms of a cross-platform application. Each of these problems were formulated in a practical scenario, where they are most relevant and real world subjects were chosen for experimentation. I have performed experiments in such realistic scenarios to provide a strong evidence to support my thesis. My work represents an important tool to help developers effectively and efficiently deal with cross-platform testing and maintenance.

7.2 Future work

In the future, as new platforms keep emerging, it is likely that cross-platform testing and maintenance will continue to be a major component of software development. Through this thesis, I have identified several challenges as a motivation and foundation that future work can address.

Cross-platform Issue Detection: One possible direction for future work that relates to issue detection, is to investigate techniques that can automatically eliminate the cross-platform issues identified by our approach. In the case of XBIs, such elimination can be performed through browser-specific web page repairs. Another possible direction is to identify inconsistencies in applications by comparing them to different variants of the application on different platforms (e.g., desktop, web, and mobile variants of web applications).

Identifying Cross-platform Missing Features: In our work, we used the network level abstraction for performing feature matching. For applications that do not heavily rely on the network, a different abstraction might be needed, which would present different challenges for performing feature matching. Another possible direction is to extend the results of feature matching to applications for uncovering the behavioral aspects across different platform front-ends, such as web and mobile (native) versions of an application.

Migrating Test Cases Across Platforms: Test migration was the most challenging problem addressed in this thesis. Our preliminary evaluation presented multiple challenges that need to be addressed in future work. The most important challenge is in the development of the *guided model generator*, which needs more sophistication to target specific differences in the cross-platform context. These requirements are

summarized as follows: 1) Platform-specific functionality, such as the flows occurring through the "back" button on Android, should be handled in the design of the model generator. 2) There should be a provision to specify the behavior of custom widgets, so that the model generator can interact with such widgets intelligently. 3) The model generator should be coupled with mechanisms to sandbox the state of the application, so that it can avoid side-effects while it is trying to learn the model.

In addition to the improvements to the model generator, we also reported the challenges in the migration of the test oracle, especially while migrating complex assertions. Migrating such assertions requires the use of deeper knowledge of the program semantics through analysis and reasoning. Another related direction is to migrate other artifacts, such as documentation, across platforms. This poses further challenges in understanding such artifacts through natural language processing techniques and correlating this information with matched entities across platforms for artifact migration.

7.3 Merit

The work in this thesis concerns the foundations of software engineering and has specific focus on improving cross-platform testing and maintenance. The techniques developed as a part of this can be used in many scenarios, where two software components have a high-level functional similarity, without expecting the unlikely notion of program equivalence. Since cross-platform applications are relevant and are likely to have more adoption with the introduction of modern computing platforms, I believe that there is scope of extending my work to other platforms than those considered in this thesis.

The techniques presented in this thesis extend the state of art in software testing and maintenance, and do so automatically by leveraging the differential testing

scenario of multi-platform systems. Hence, the improvements introduced by the techniques do not require much effort from the developer and are not intrusive. In addition, the preliminary evaluation of the techniques on real world applications indicates that the approach can reduce the cost and difficulty of performing these testing and maintenance tasks. Together the techniques address multiple issues, which arise in the cross-platform context and provide an over-arching approach to apply this framework to address different testing and maintenance tasks.

REFERENCES

[1] "[Plugin: Wordpress mobile pack] adding media or tags." http://wordpress.org/support/topic/plugin-wordpress-mobile-pack-adding-media-or-tags, 2010.

[2] "[Plugin: Wordpress mobile pack] allow author access to mobile admin." http://wordpress.org/support/topic/plugin-wordpress-mobile-pack-allow-author-access-to-mobile-admin, 2012.

[3] "What are the product management best practices for building one product across multiple platforms." http://qr.ae/Qlx1F, July 2012.

[4] "Wordpress mobile pack." http://wordpress.org/plugins/wordpress-mobile-pack/, May 2012.

[5] "Android fragmentation visualized." http://opensignal.com/reports/2014/android-fragmentation/, August 2014.

[6] "Acid Tests - The Web Standards Project." http://www.acidtests.org.

[7] Alshahwan, N. and Harman, M., "Automated session data repair for web application regression testing," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, (Washington, DC, USA), pp. 298–307, IEEE Computer Society, 2008.

[8] Benson, G., "Tandem repeats finder: a program to analyze dna sequences.," *Nucleic acids research*, vol. 27, no. 2, p. 573, 1999.

[9] Berners-Lee, T., Masinter, L., McCahill, M., and others, "Uniform resource locators (url)," 1994.

[10] Bradski, G. and Kaehler, A., *Learning OpenCV*. O'Reilly Media, September 2008.

[11] Browserbite, "Cross browser testing with computer vision." http://app.browserbite.com/.

[12] calleho, "Theme: imobile for iphone and ipad." http://galleryproject.org/node/101768.

[13] Chapman, C., "Review of cross-browser testing tools.." http://www.smashingmagazine.com/2011/08/07/a-dozen-cross-browser-testing-tools/, August 2011.

[14] Chen, J., "Formal Modelling of Java GUI Event Handling," in *Formal Methods and Software Engineering* (George, C. and Miao, H., eds.), vol. 2495 of *Lecture Notes in Computer Science*, pp. 359–370, Springer Berlin Heidelberg, 2002.

[15] Choi, W., Necula, G., and Sen, K., "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, (New York, NY, USA), pp. 623–640, ACM, 2013.

[16] Clarke, L., "A system to generate test data and symbolically execute programs," *Software Engineering, IEEE Transactions on*, vol. SE-2, pp. 215–222, Sept 1976.

[17] Clause, J., Li, W., and Orso, A., "Dytan: A Generic Dynamic Taint Analysis Framework," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, (London, UK), pp. 196–206, July 2007.

[18] Clausen, J., "Branch and bound algorithms-principles and examples," 1999.

[19] Dallmeier, V., Burger, M., Orth, T., and Zeller, A., "WebMate: a tool for testing web 2.0 applications," in *Proceedings of the Workshop on JavaScript Tools (JSTools)*, pp. 11–15, ACM, June 2012.

[20] Daniel, B., Jagannath, V., Dig, D., and Marinov, D., "ReAssert: Suggesting Repairs for Broken Unit Tests," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 433–444, IEEE Computer Society, 2009.

[21] Di Lucca, G. A., Fasolino, A. R., and Tramontana, P., "Reverse engineering web applications: the ware approach," *Journal of Software Maintenance and Evolution*, vol. 16, pp. 71–101, Jan. 2004.

[22] Dobolyi, K. and Weimer, W., "Harnessing web-based application similarities to aid in regression testing," in *Proceedings of the 20th IEEE international conference on software reliability engineering*, ISSRE'09, (Piscataway, NJ, USA), pp. 71–80, IEEE Press, 2009.

[23] DudaMobile, "Mobile websites made easy." `http://www.dudamobile.com/`, September 2013.

[24] Eaton, C. and Memon, A. M., "An empirical approach to evaluating web application compliance across diverse client platform configurations," *International Journal of Web Engineering and Technology*, vol. 3, pp. 227–253, January 2007.

[25] Fielding, R. T. and Taylor, R. N., "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, pp. 115–150, May 2002.

[26] Fling, B., *Mobile Design and Development: Practical concepts and techniques for creating mobile sites and web apps*, ch. 11. O'Reilly Media, 2009.

[27] Gokhale, A., Ganapathy, V., and Padmanaban, Y., "Inferring likely mappings between apis," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 82–91, IEEE Press, 2013.

[28] Google, "Youtube mobile gets a kick start." `http://youtube-global.blogspot.com/2010/07/youtube-mobile-gets-kick-start.html`, 2010.

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

[30] Grosskurth, A. and Godfrey, M. W., "A reference architecture for web browsers," *21st IEEE International Conference on Software Maintenance*, pp. 661–664, September 2005.

[31] Haldar, V., Chandra, D., and Franz, M., "Dynamic taint propagation for java," in *Computer Security Applications Conference, 21st Annual*, pp. 9–pp, IEEE, 2005.

[32] Hao, S., Liu, B., Nath, S., Halfond, W. G., and Govindan, R., "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, (New York, NY, USA), pp. 204–217, ACM, 2014.

[33] Jaccard, P., "Distribution de la flore alpine dans le bassin des drouces et dans quelques regions voisines," in *Bulletin de la Socit Vaudoise des Sciences Naturelles*, vol. 37, p. 241272, 1901.

[34] Jacobson, G. and Vo, K.-P., "Heaviest increasing/common subsequence problems," in *Combinatorial Pattern Matching*, vol. 644 of *Lecture Notes in Computer Science*, pp. 52–66, Springer Berlin Heidelberg, 1992.

[35] jQuery Mobile, "Touch-optimized web framework for smartphones & tablets." `http://jquerymobile.com/`, September 2013.

[36] Kervinen, A., Maunumaa, M., Pääkkönen, T., and Katara, M., "Model-Based Testing Through a GUI," in *Proceedings of the 5th International Conference on Formal Approaches to Software Testing*, FATES'05, (Berlin, Heidelberg), pp. 16–31, Springer-Verlag, 2006.

[37] Knowledge, C., "FILExt: The file extension source." `http://filext.com/`.

[38] Kuhn, H. W., "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[39] Land, A. H. and Doig, A. G., "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, vol. 28, pp. 497–520, 1960.

[40] Levenshtein, V., "Binary codes capable of correcting spurious insertions and deletions of ones," *Problems of Information Transmission*, vol. 1, pp. 8–17, 1965.

[41] Ma, S., "10 ways mobile sites are different from desktop web sites." `http://www.uxmatters.com/mt/archives/2011/03/10-ways-mobile-sites-are-different-from-desktop-web-sites.php`, 2011.

[42] Manning, C. D., Raghavan, P., and Schütze, H., *Introduction to information retrieval*, vol. 1. Cambridge University Press Cambridge, 2008.

[43] Memon, A. M., "An event-flow model of gui-based applications for testing: Research articles," *Softw. Test. Verif. Reliab.*, vol. 17, pp. 137–157, Sept. 2007.

[44] Memon, A. M. and Soffa, M. L., "Regression testing of guis," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, (New York, NY, USA), pp. 118–127, ACM, 2003.

[45] Mesbah, A. and Prasad, M. R., "Automated cross-browser compatibility testing," in *Proceeding of the 33rd International Conference on Software Engineering (ICSE)*, pp. 561–570, ACM, May 2011.

[46] Mesbah, A., van Deursen, A., and Lenselink, S., "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, vol. 6, pp. 3:1–3:30, March 2012.

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

[48] Mobify, "Adaptive platform for responsive websites." `http://www.mobify.com/`, September 2013.

[49] Munkres, J., "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. pp. 32–38, 1957.

[50] Robillard, M., Bodden, E., Kawrykow, D., Mezini, M., and Ratchford, T., "Automated api property inference techniques," *Software Engineering, IEEE Transactions on*, vol. 39, no. 5, pp. 613–637, 2013.

[51] Roy Choudhary, S., Prasad, M. R., and Orso, A., "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Proceedings of the IEEE Fifth International Conference on*

Software Testing, Verification, and Validation (ICST), pp. 171–180, IEEE, April 2012.

[52] Roy Choudhary, S., Prasad, M. R., and Orso, A., "X-PERT: Accurate identification of cross-browser issues in web applications," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 702–711, IEEE Press, 2013.

[53] Roy Choudhary, S., Versee, H., and Orso, A., "WebDiff: Automated identification of cross-browser issues in web applications," in *Proceeding of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, pp. 1–10, IEEE, September 2010.

[54] Roy Choudhary, S., Zhao, D., Versee, H., and Orso, A., "WATER: Web Application TEst Repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, (New York, NY, USA), pp. 24–29, ACM, 2011.

[55] Safairis, I., "15 useful tools for cross browser compatibility test.." `http://wptidbits.com/webs/15-useful-tools-for-cross-browser-compatibility-test/`, March 2011.

[56] Schur, M., Roth, A., and Zeller, A., "Mining behavior models from enterprise web applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 422–432, ACM, 2013.

[57] Sebatian Elbaum, Gregg Rothermal, S. K. and II, M. F., "Leveraging user-session data to support web application testing," *IEEE Transactions on Software Engineering*, vol. 31, pp. 187–202, March 2005.

[58] Sencha Touch, "Build mobile web apps with html5." `http://www.sencha.com/products/touch`, September 2013.

[59] Shevchik, L., "Mobile device fragmentation: Its only going to get worse." `http://blog.newrelic.com/2013/05/23/mobile-device-fragmentation-its-only-going-to-get-worse/`, May 2013.

[60] Stackoverflow. `http://data.stackexchange.com/stackoverflow/query/77488/posts-for-cross-browser-issues`, August 2012.

[61] Tamm, M., "Fighting layout bugs." `http://code.google.com/p/fighting-layout-bugs/`, October 2009.

[62] "test262 - ECMAScript." `http://test262.ecmascript.org/`.

[63] Tsai, W.-T., Volovik, D., and Keefe, T., "Automated test case generation for programs specified by relational algebra queries," *Software Engineering, IEEE Transactions on*, vol. 16, pp. 316–324, Mar 1990.

[64] Twitter, "Overhauling mobile.twitter.com from the ground up." https://blog.twitter.com/2012/overhauling-mobiletwittercom-ground, 2012.

[65] Twitter Bootstrap, "Sleek, intuitive, and powerful front-end framework for faster and easier web development.." http://getbootstrap.com/, September 2013.

[66] Willamson, L., "A mobile application development primer: A guide for enterprise teams working on mobile application projects," *IBM Software: Thought Leadership White Paper*, 2013.

[67] World Wide Web Consortium, "Mobile web best practices 1.0." http://www.w3.org/TR/2008/REC-mobile-bp-20080729/, July 2008.

[68] Zhong, H., Thummalapenta, S., Xie, T., Zhang, L., and Wang, Q., "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, (New York, NY, USA), pp. 195–204, ACM, 2010.