Cross-Platform Feature Matching for Web Applications

Shauvik Roy Choudhary[*], Mukul R. Prasad[†], Alessandro Orso[*]

[*]College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
{shauvik | orso}@cc.gatech.edu

[†]Software Systems Innovation Group
Fujitsu Laboratories of America
Sunnyvale, CA, USA
mukul@us.fujitsu.com

ABSTRACT

With the emergence of new computing platforms, software written for traditional platforms is being re-targeted to reach the users on these new platforms. In particular, 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. Because the differences between desktop and mobile versions of a web application are not only cosmetic, but can also include substantial rewrites of key components, it is not uncommon for these different versions to provide different sets of features. Whereas some of these differences are intentional, such as the addition of location-based features on mobile devices, others are not and can negatively affect the user experience, as confirmed by numerous user reports and complaints. Unfortunately, checking and maintaining the consistency of different versions of an application by hand is not only time consuming, but also error prone. To address this problem, and help developers in this difficult task, we propose an automated technique for matching features across different versions of a multi-platform web application. We implemented our technique in a tool, called FMAP, and used it to perform a preliminary empirical evaluation on nine real-world multi-platform web applications. The results of our evaluation are promising, as FMAP was able to correctly identify missing features between desktop and mobile versions of the web applications considered, as confirmed by our analysis of user reports and software fixes for these applications.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*portability, reverse engineering*

General Terms

Software Maintenance, Software Testing

Keywords

Cross-Platform, Mobile Web

ISSTA '14, July 21-25, 2014, San Jose, CA, USA

1 INTRODUCTION

Today people use software on a variety of platforms, including desktop computers, mobile devices such as smartphones and tablets, and even wearable embedded computing devices [15, 11]. In fact, desktop computers are rapidly being supplanted by mobile devices as the preferred means of accessing Internet content. Case in point, the market research firm IDC predicts that, by the year 2015, more users will be accessing the Internet from mobile devices than from their personal computers [35]. This move to mobile platforms has been fueled, in part, by the increasing computing power of modern mobile devices, coupled with their rich interactive user interface, portability, and convenience.

Because of this increasing prevalence of mobile devices and platforms, most companies whose business largely depends on web presence, build versions of their existing web applications customized for mobile devices, so as 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 [36]. Developers thus commonly re-target their web applications, sometimes substantially, to make them more suitable for mobile platforms [12].

In spite of the inherent differences between desktop and mobile platforms, and the resulting differences between desktop and mobile versions of a web application, 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 [37], 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 [14] and Twitter [33] now follow this guideline, and Figure 1 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, customized to suit the specific characteristics of the different platforms,

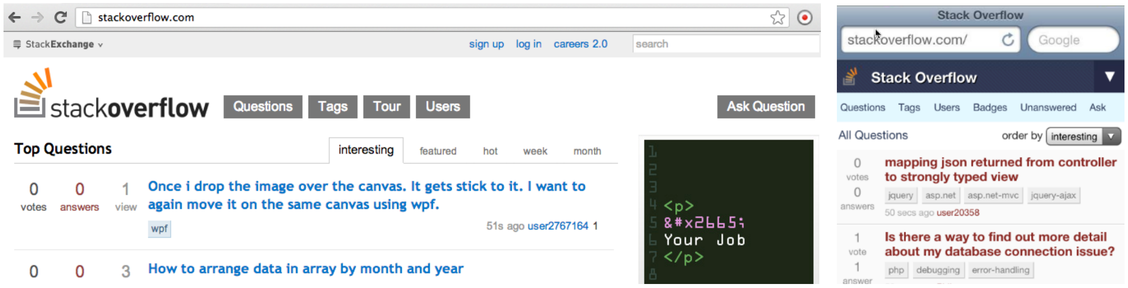

Figure 1: StackOverflow.com on desktop (left) and mobile (right).

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 [12].

Despite the existence of several libraries and frameworks for helping with this task (*e.g.,* jQuery Mobile [19], Twitter Bootstrap [34], or Sencha [32]), and even tools for migrating existing web applications to mobile-friendly versions (*e.g.,* Mobify [26] or Dudamobile [10]), 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 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 5).

To help developers address the challenges associated with developing multi-platform web applications, in this paper we propose a technique to automatically match features across different versions of such applications. To do so, we first introduce the notion of consistency between different versions of a web application and define it in terms of correspondence among *features* supported by the different versions. We then propose a novel technique for matching features across platform-specific versions of a given 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 occurs 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) abstract each trace into 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.

We implemented our approach in a tool called FMAP and used FMAP to perform a preliminary evaluation of our technique on nine real-world multi-platform web applications. The results of our evaluation are promising and motivate further research in this direction. FMAP correctly identified cases of missing features between desktop and mobile versions of the web applications considered, including cases that were reported by users and cases that were later fixed by developers. Moreover, FMAP was able to also handle complex cases in which platform-specific versions of the web application had a totally different look and feel. This confirms our intuition that client-server communication can be used to characterize web-application features even in the presence of significantly different front-ends.

The main contributions of this paper 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, as a peer-reviewed artifact of this paper (`http://gatech.github.io/fmap`).
- An empirical evaluation of our technique on nine real-world multi-platform web applications.

2 WEB APP FEATURE MATCHING

2.1 Web Applications

Web applications follow a distributed, client-server computing architecture. They are hosted on a particular web server, connected to the internet and are accessed by end users on any browser of their choice, running on a variety of desktop and mobile platforms. Web standards enable a developer to write once and make the application available on such diverse platforms. In particular, the user supplies the URL to the web browser and interacts with the web application inside it. Behind the scenes, the browser makes several requests to the server to fetch resources required to render the application for the user. These resources are essentially of four types: 1) Data, in the form of HTML or XML files, 2) Style information, such as cascading style sheets, 3) Client-side code, in the form of JavaScript, and 4) Binary files, such

Figure 2: MakeMyPost.com Web Application for Desktop (top) and Mobile (bottom) Browsers

as images, audio and video. Web browsers on different platforms follow standards set by W3C [4] to use these resources for rendering the web application to the user.

2.2 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. Our example, *MakeMyPost.com*, as shown in Figure 2, is a content management system, and provides different front-ends for the desktop and mobile platforms.

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, the alignment of the text box, and the corresponding labels are different. Further, the "Remember me" check-box, the "Forgot pass" button, and their corresponding functionality, are not provided on the mobile view. The next "Home" screen allows the user to create a new post and is somewhat different across the platforms: 1) Navigation tabs present on the desktop version are replaced by a dropdown on mobile, 2) Radio buttons to select the "Post Type" are missing on the mobile screen, and, 3) The appearance of the buttons is different across the platforms.

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 [29] would not work in this context.

Now, let us consider the client-server network communication originating from both application versions for the use-case of creating a post. In this case, 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 Figure 3. These requests are largely similar across the platform versions, 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 paper, 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.

2.3 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-side of web applications, and may carry different meanings in other contexts.

Definition 1 (Service). *A service is an atomic functionality offered by the web server to all clients, which may be invoked, potentially under different contexts, by such clients.*

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

Definition 2 (Request). *A request is a call made from the client browser to the server, to obtain display resources, exercise a service, or to navigate the user-interface to gain access to a particular service.*

Definition 3 (Response). *A response is the reaction of the server to a request from the client.*

Definition 4 (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 3 shows a trace from the desktop and mobile versions of *MakeMyPost.com*, corresponding to the use-case of logging in and creating a post. 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 obtain display resources or navigate the user-interface.

Definition 5 (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 3 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).

```
 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'
```

```
 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'
```

Figure 3: Network trace from MakeMyPost.com on desktop (left) and mobile (right).

Definition 6 (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. In 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, thus correspond to the same action.

Definition 7 (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 3 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 [24]. 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 8 (Feature Mapping Problem). *Given two versions $\mathcal{W}_1$ and $\mathcal{W}_2$ of a web application, 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 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 invocations and hence identification of features needs to be performed by indirectly leveraging other information.

Our technique, developed in Section 3, presents our solution to these challenges.

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 2.3, 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 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 [24, 30] or event-based models [23] would not be particularly useful in th is context.

Figure 4 presents a high-level view of our proposed 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 2.3. 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 alpha-

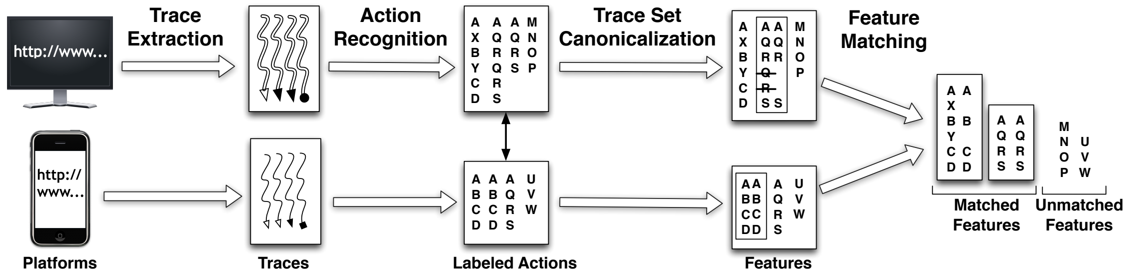

Figure 4: High-level overview of FMAP.

bet 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 proposed technique, using our motivating example from Section 2.2 to illustrate key concepts and operations.

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 3 contains several examples of such request-response pairs.

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. Algorithm 1 presents *RecognizeActions*, the main procedure of this step. It takes a set of network traces from the two platforms, and returns a set of labeled actions. The key operations 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 operations are explained in detail below.

Algorithm 1: 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
1  begin
2      C_d ← ClusterActions(𝕋_d)
3      C_m ← ClusterActions(𝕋_m)
       // Action Mapping
4      Map ← { }
5      foreach c1 ∈ C_d do
6          foreach c2 ∈ C_m do
7              if isSimilar(c1, c2) then
8                  if c1 ∈ Map or c2 ∈ Map then
9                      c1 ← c1 ∪ Map.remove(c1)
10                     c2 ← c2 ∪ Map.remove(c2)
11                 Map.add(c1 ↦ c2)
12     𝔸_d ← [], 𝔸_m ← []
13     foreach (c1, c2) ∈ Map do
14         action ← getNewSymbol()
15         𝔸_d.assign(c1, action)
16         𝔸_m.assign(c2, action)
17     foreach c1 ∈ C_d and c1.action == null do
18         𝔸_d.assign(c1, getNewSymbol())
19     foreach c2 ∈ C_m and c2.action == null do
20         𝔸_m.assign(c2, getNewSymbol())
21     return 𝔸_d, 𝔸_m

   /* ClusterActions */
   Input  : 𝕋: Set of traces
   Output: ℂ: Cluster of actions
22 begin
23     𝒦 ← TraceSimplify(𝕋)
       // Level 1 Clustering
24     L1Cluster ← SimpleCluster(𝕋, url_path_equals)
       // Level 2 Clustering
25     L2Cluster ← { }
26     JD ← {JaccardDistance(k_1, k_2) | k_1, k_2 ∈ 𝒦}
27     underCluster ← split(L1Cluster, size == 1)
28     overCluster ← split(L1Cluster, size > 1)
29     L2Cluster.add(AggloCluster(underCluster, JD, (<, t_1)))
30     foreach c ∈ overCluster do
31         L2Cluster.add(AggloCluster(c, JD, (>, t_2)))
32     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}
33 begin
34     𝒦 ← ( )
35     foreach 𝒯 ∈ 𝕋 do
36         foreach ⟨request, response⟩ ∈ 𝒯 do
37             while isRedirect(response.code) do
38                 response ← followRedirect(response)
39             if isCodeOrData(response.type) then
40                 k ← getKeyws(request.path, request.qs)
41                 𝒦.add(k)
42     return 𝒦
```

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 requests 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 3), 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 [6]. 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 [20]. The extracted words are further simplified by converting them to their lemmas by using Lemmatisation [22]. 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')]`.

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 [22], 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 [17] 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.

3.2.3 Action Mapping

To achieve the overall goal of feature mapping, in this step, similar actions across the desktop and mobile are grouped together. 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.

3.3 Trace Set Canonicalization

The goal of this step is to identify and cluster together 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.

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, e.g., creating multiple blog posts, in the context of our motivating example.

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 [5]. 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 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 `(AQRQRS, AQRS)` Figure 4. The technique will first replace the tandem repeat of subsequence `QR` in the first trace with a single `QR`. 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.

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

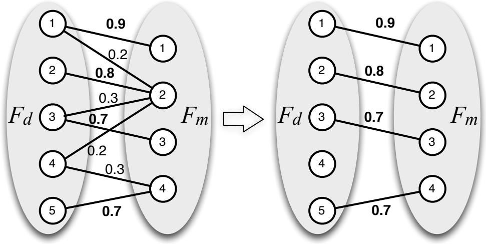

Figure 5: Bipartite graph of features

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 $G = (V, E)$, 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 [21, 27]. 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 5 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 edge. On the right side of the figure is the solution to the MWBM problem where only 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 $[(1, 1), (2, 2), (3, 3), (5, 4)]$ for the example. The figure also shows features that were unmatched. (i.e., feature 4 in F_d in the 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.

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 7 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 the number of 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 containing a, out of all feature instances (F), and $|F|$ denotes the cardinality of feature instances. Thus, if an action occurs in all features its weight is zero. However if it occurs in fewer features it is assigned a weight closer to 1. Once these weights are assigned, they are used to compute the heaviest common subsequence match from respective traces.

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 [18]. The HCS problem can be defined formally using the following recursive formula:

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

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 respectively. 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 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 proposed 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

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

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 service 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.

4.1 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 allows the user to select if they want to use the desktop or mobile browser. To emulate the mobile browser, it alters the HTTP user agent string in the network requests to iPhone 5 browser agent. The network request-response information is captured through the browser's debugger interface and the trace generated is saved to a file, which is named with the use-case name obtained from the user. The extension also saves screenshots of the visited web application screens to facilitate our evaluation.

The next component of FMAP is written in Python and implements the action recognition, feature identification, and feature matching steps of the technique. To detect the several inflected forms of words as one, the words are reduced to their root forms by using the WordNet lemmatizer [25] in the Python natural language toolkit (`http://nltk.org`). All metrics were computed by using the corresponding methods from the `nltk.metrics` package. We used the open source python library, `munkres`, suitably modified to handle floating point profits, for solving the MWBM problem.

4.2 Subjects

To perform a meaningful evaluation of our technique, we selected nine web applications whose mobile and desktop versions appear to be quite different (see Table 1). The first six subjects are popular open source web applications obtained from `http://ohloh.net`: `wordpress` v3.6, a web blogging tool; `drupal` v7.23, a content management app; `phpbb` v3.0, a bulletin board; `roundcube` v0.9.4, an email client; `elgg` v1.8.16, a social networking app; and `gallery` v3.0.9, 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 v1.2.5, nokia mobile theme v6.x-1.3 for drupal, artodia mobile style v3.4 for phpbb, mobilecube theme v3.0.0 for roundcube, elgg mobile module v2.0, and imobile theme v2.7 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 across platforms, and were quite different from our open source applications.

4.3 Protocol

To collect the experimental data for our evaluation, five graduate students were recruited. 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 FMAP's 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.

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 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 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 [22], 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 1, 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 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 2 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.

Table 1: 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

Table 2: 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**

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 [8, 16] 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 performs 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 reported unmatched by FMAP and are potentially missing. Our analysis of this result revealed several missing features as reported in Table 2, 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 in the application [3]. Specifically, users complained about not being able to upload media, assign categories to posts, or administer their blog from mobile [1, 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 [7] 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 reported missing features, which were related to the viewing or editing the profile details of the logged in user. Although, Twitter's private support requests are not open, 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 favorite 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 good evidence of the effectiveness of FMAP in matching as well as finding missing features.

5.1 Limitations

Since our proposed technique is the first solution to the feature mapping problem, albeit a promising one, it has certain limitations. We discuss these in the following while noting that many of these can be mitigated by further research in this area.

Trace collection: Since the traces used in our current experiments were captured by student volunteers, recruited by us, a valid concern might be the dependence of our results on the choice of these traces. A specific concern might be a bias towards selecting similar use-cases and traces across the two platforms, while traces encountered in real-world scenarios might be very different or consist of complicated interleavings of features. To mitigate such bias we instructed the volunteers to independently select traces from different platforms, choosing different volunteers to work with different platforms, where possible. Further, our technique includes specific steps, such as canonicalization of the traces, to abstract away such differences. However, future work, supported by more general trace collection strategies is needed to fully address this concern.

Omitted vs. missing features: It is difficult to automatically distinguish between erroneously missed features and ones that have been intentionally omitted by the developers, on certain platforms. Currently FMAP reports both of them as missing features. However, the technique can be easily modified to accept a list of intentionally omitted features and to ignore such intentional feature discrepancies while reporting the erroneously missed features. For our subjects, we were indeed able to find instances where ostensibly omitted features, reported by FMAP as missing features, were not acceptable to some users of the applications. However, understanding the distribution of erroneous versus intentionally omitted features, in web applications in the wild, and modifying the technique to respect this distinction, needs to be an important aspect of future work.

Generality of conclusions: The success of our technique relates to the inter-platform similarity of the web application, and specifically, on the web application's design following the One Web principle. To avoid a selection bias, we tried to pick a diverse and challenging set of subjects: popular web applications, which had dynamic features and demonstrated a clear difference in appearance across platforms. The action clustering step in Algorithm 1 relies on two thresholds, $t1$ and $t2$. We performed a sensitivity study by independently varying each threshold by ± 0.1, and observed that it did not significantly change the clustering result. However, as with all studies, the external validity of our conclusions will increase with more subjects and experimentation.

6 RELATED WORK

Cross-browser testing: The objective of cross-browser testing is to detect discrepancies between the renderings of a given web application under different web browsers, typically on the same platform. Most prior work on this problem, including the authors' recent paper on the X-PERT tool [29] is predicated on the single platform assumption. In this setting, both presentation and function of the web application is expected to be largely identical across different browsers. Any differences, where present, are subtle, and typically confined to the layout of individual web pages. By contrast, the desktop and mobile versions of a web application can differ substantially in their look, feel, and even their workflow. Thus, the feature mapping problem addressed in this work is about discovering very deep-seated, fundamental similarities in the functionality of the two versions, in the face of largely dissimilar looking presentation and behavior. In that sense, it is, in substance, quite the opposite of what is solved by cross-browser testing techniques.

Inferring API migration mappings: There is a body of research [13, 38, 28] 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 [13] 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 similar behavior at the client-server interface. No such interface exists or can be exploited by API matching techniques.

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 include tools that generate a model of the web application [24, 30, 9, 31]. 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 such techniques, or derived from some other source of manually or automatically generated test-cases.

7 CONCLUSION

In this paper, we introduced and defined the problem of missing features in web applications, applications that are developed in multiple platform- specific versions (*e.g.,* desktop or mobile). We proposed a novel technique to address this problem and presented its implementation in a tool called FMAP. Our technique analyzes the client-server communication of different versions of a web application to match features across platforms. Our preliminary evaluation of FMAP, performed on nine real-world multi-platform web applications, is promising. FMAP was able to correctly identify 58 true missing features in the web applications considered. Moreover, 40 out of the issues identified were confirmed by real user reports or by examining software fixes to the application. In future work, we plan to investigate extensions of our technique to aid software testing and maintenance tasks. One direction is to use feature mapping to uncover behavioral differences across different platform front-ends.

8 ACKNOWLEDGEMENTS

This was work supported in part by NSF awards CCF-1161821, CNS-1117167, and CCF-0964647 to Georgia Tech, and by a research contract with Fujitsu Labs of America. We also thank the five graduate students from Georgia Tech for their help with trace extraction.

9 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] Wordpress mobile pack. http://wordpress.org/plugins/wordpress-mobile-pack/, May 2012.

[4] The World Wide Web Consortium (W3C). http://www.w3.org/, Jan 2013.

[5] G. Benson. Tandem repeats finder: a program to analyze dna sequences. *Nucleic acids research*, 27(2):573, 1999.

[6] T. Berners-Lee, L. Masinter, M. McCahill, et al. Uniform resource locators (url). 1994.

[7] calleh. Theme: iMobile for iPhone and iPad. http://galleryproject.org/node/101768.

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

[9] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. Reverse engineering web applications: the ware approach. *Journal of Software Maintenance and Evolution*, 16(1-2):71–101, Jan. 2004.

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

[11] Fitbit. `http://www.fitbit.com/`.

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

[13] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring Likely Mappings between APIs. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 82–91, Piscataway, NJ, USA, 2013. IEEE Press.

[14] Google. YouTube Mobile gets a kick start. `http://youtube-global.blogspot.com/2010/07/youtube-mobile-gets-kick-start.html`, 2010.

[15] Google Glass. `http://www.google.com/glass/start/`.

[16] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Computer Security Applications Conference, 21st Annual*, pages 9–pp. IEEE, 2005.

[17] P. Jaccard. Distribution de la flore alpine dans le bassin des drouces et dans quelques regions voisines. In *Bulletin de la Société Vaudoise des Sciences Naturelles*, volume 37, page 241-272. 1901.

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

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

[20] C. Knowledge. FILExt: The file extension source. http://filext.com/.

[21] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[22] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*, volume 1. Cambridge University Press Cambridge, 2008.

[23] A. M. Memon. An event-flow model of GUI-based applications for testing: Research Articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, Sept. 2007.

[24] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web*, 6(1):3:1–3:30, March 2012.

[25] G. A. Miller. Wordnet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.

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

[27] J. Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):pp. 32–38, 1957.

[28] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.

[29] S. Roy Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 702–711, Piscataway, NJ, USA, 2013. IEEE Press.

[30] M. Schur, A. Roth, and A. Zeller. Mining Behavior Models from Enterprise Web Applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 422–432, New York, NY, USA, 2013. ACM.

[31] S. K. Sebatian Elbaum, Gregg Rothermal and M. F. II. Leveraging User-Session Data to Support Web Application Testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, March 2005.

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

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

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

[35] K. Weide. Worldwide new media market model 1h12 highlights: Internet becomes ever more mobile, ever less pc based. Technical Report 237459, International Data Corporation, Oct 2012.

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

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

[38] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API Mapping for Language Migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 195–204, New York, NY, USA, 2010. ACM.