GUI Test Transfer from Web to Android

Jun-Wei Lin and Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
{junwel1, malek}@uci.edu

***Abstract*—GUI testing is important for examining the end-to-end workflows and usability of GUI-based software. To reduce the manual effort of writing GUI tests, recent research has explored the potential of automatically reusing GUI tests by transferring them across similar applications. However, what is missing from the prior work is that such transfer may be required for apps available on different platforms. In particular, both web and Android are dominant platforms on which many organizations provide their software services. At the state-of-the-practice, even if the web and Android versions of an app provisioned by an organization substantially share the functionality, the developers have to manually write separate sets of tests for each version. This paper proposes TRANSDROID, an automated tool that transfers GUI tests from a web app to its Android counterpart. Evaluation of TRANSDROID on real-world web and Android apps corroborates its effectiveness by achieving 77% success rate among the attempted transfers, along with 82% precision and 99% recall in the mapping of the GUI events and oracles.**

***Index Terms*—Test Transfer, Test Reuse, Cross-Platform Testing, GUI Testing**

I. INTRODUCTION

Usage-based GUI testing aims to cover the use-case scenarios of the software under test. Developers typically prefer usage-based GUI testing to other forms of GUI testing (e.g., crawling) that are use-case agnostic and simply aim for higher code coverage [1], [2]. Usage-based GUI testing provides the developers with actionable information that allows them to properly recreate the failures and debug their programs. However, this form of testing is tedious and time-consuming, since it often involves substantial manual effort of writing the test cases from scratch.

To reduce the manual effort of writing usage-based GUI tests, recent research has explored the possibility of reusing GUI tests by automatically transferring them across similar applications (apps) within a platform [3]–[8]. By platform, we mean a particular computing domain, such as mobile, web, and desktop. A key insight guiding these efforts is that the GUI widgets of different apps providing the same functionality are semantically similar. As a result, it is possible to automatically generate a usage-based GUI test for a *target app* by reusing the test of a *source app*, provided that (1) both apps share the same feature (functionality); and (2) the correct mapping of the widgets between these two apps can be identified.

While the current techniques for intra-platform test transfer are promising, missing from the prior work is that such transfer may be required for apps on different platforms. In fact, many organizations provide their software services on multiple platforms. Case in point, among the top 50 most visited websites in the United States [9], 80% of them also provide native mobile apps for their users. Another example is WordPress [10], one of the most popular content management systems, which can be accessed via web browsers, mobile apps (Android and iOS), and desktop apps (Windows, MacOS, and Linux). At the state-of-the-practice, despite substantial overlap among several versions of an app provisioned by an organization and intended for execution on different platforms, developers have to manually write separate sets of tests for each version of app. We believe automated test transfer presents a promising solution in such settings, yet has never been explored in the past.

There are two main challenges in cross-platform test transfer that prior work has not addressed. The first is *incompatible actions*. Event synthesis is a necessary process for test transfer, in which appropriate actions such as a *click* are determined for the identified GUI widgets in the target app to compose executable events. This synthesis is guided by both the actions performed by the source test and the type of target widgets. While GUI-based apps share certain common actions such as *click* and *text input*, different platforms usually provide additional unique actions to optimize the user experience. As a result, if the source actions are platform-specific and not supported on the target platform, current techniques are not able to finish the transfer. For example, *mouseOver* is a common action in web testing for sub-menu exploration, but its corresponding action on Android is undefined.

The second challenge in cross-platform test transfer is *unclear widget context*. A core process in test transfer is to search and map the GUI widgets between the source and target apps. For example, what is most similar to the "Sign Up" button in the source app can be the "Register" button in the target app. In this case, a source GUI event clicking the "Sign Up" button can be transferred to a target event clicking the "Register" button. In prior work, the similarity between widgets are determined by their context, such as text values (e.g., "Register") and types (e.g., `Android.widget.Button`).

As part of the context, type information is important for the search and mapping of the widgets. For instance, if the source widget is an `Android.widget.Button`, the most similar target widget is likely also an `Android.widget.Button` (or at least a *clickable*). Nevertheless, such context may be missing or ambiguous when the transfer crosses platform boundaries. Take GUI widgets in web apps, i.e., HTML tags, as an example. There are two main categories of HTML tags: specific tags (e.g., `<a>`, `<textarea>`, and `<li>`) and generic tags (e.g., `<span>` and `<div>`). A characteristic

of web apps is that, through registered JavaScript event handlers, the behavior of widgets can be easily changed or assigned. For example, developers can change the behavior of a `<textarea>` from *editable* to *clickable*. Furthermore, it is even more common to assign arbitrary behaviors to generic tags like `<span>`. In turn, context of source widgets on web may provide no or even wrong hints for the search and mapping of target widgets on, for instance, a mobile platform, like Android.

In this paper, we propose TRANSDROID, an automated tool that addresses the aforementioned challenges in the context of web-to-Android test transfer. In other words, TRANSDROID transfers GUI tests from a web app to its Android counterpart. The reason for this implementation choice is the fact that the Internet era precedes the smartphone era [11], [12], and there are a large number of organizations developing their web app prior to their mobile app. Typical examples include WordPress [10], Wikipedia [13], Twitter [14], and Zoom [15]. As a result, we believe many organizations may benefit from TRANSDROID, allowing the tests created for their web app to be reused for their mobile app. Nevertheless, it should be noted that the aforementioned challenges are shared in all types of cross-platform test transfer, e.g., mobile-to-web, web-to-desktop, and we expect the overall approach described in this paper to have application in other domains, albeit with a different implementation to account for platform-specific differences.

TRANSDROID has several key differences from prior work. First, it includes a pre-transfer phase with customizable action transformation rules to covert incompatible actions in the source test into compatible ones with the target platform. Moreover, besides widget context, TRANSDROID considers two other types of contextual information, i.e., *screen context* and *action context*. The inclusion of additional contextual information not only helps the search and mapping of the GUI widgets with unclear widget context, but also makes TRANSDROID capable of supporting the transfer of test steps or events that have no associated GUI widgets, e.g., *jumpByURL* event that navigates to a web page by directly changing the URL field in the browser.

We evaluated TRANSDROID with 20 real-world web and Android apps and 110 test cases, including 561 GUI and oracle events for the web apps. The experimental results show that 77% of the attempted transfers were successful, along with 82% precision and 99% recall for the widget mapping.

In short, this paper makes the following contributions:

- A description of cross-platform test transfer problem and the associated challenges in the context of web-to-Android transfer.
- A novel approach to automatically transfer GUI tests from a web app to its Android counterpart. The tool implementing this approach is publicly available [16].
- An empirical evaluation on real-world apps demonstrating the effectiveness and efficiency of the proposed approach.

TABLE I: The GUI and oracle events for saving a draft in WordPress on different platforms

Source Events on Web	Target Events on Android
1. ("Posts", *mouseOver*) 2. ("All Posts", *click*)	("Posts", *click*)
3. ("Add New", *click*)	("Create a Post", *click*)
4. ("Title", (*input*, "Blog Title"))	("Title", (*input*, "Blog Title")
5. ("Content", (*input*, "Blog Content"))	("Content", (*input*, "Blog Content"))
6. ("Save Draft", *click*)	("More options", *click*) ("Save", *click*)
7. ("", (*JumpByURL*, `all-posts.php`))	("Posts", *click*)
8. ("Draft", *click*)	("Drafts", *click*)
9. ("Blog Title", *isDisplayed*)	("Blog Title", *isDisplayed*)

The rest of this paper is organized as follows. Section II provides the background for understanding this work using a motivating example. Section III provides an overview of TRANSDROID as well as the implementation details of its components. Section IV illustrates our proposed test generation algorithm. Section V presents the evaluation results. Section VI discusses the limitations of this work. The paper concludes with an overview of the related research and future work.

II. BACKGROUND AND MOTIVATING EXAMPLE

User interaction with GUI-based software is in terms of GUI events. A GUI event (w, a) consists of a widget w and an action a performed on w. Note that it is possible that there is no widget associated with a GUI event, such as the *jumpByURL* event mentioned earlier. Moreover, an action in GUI events can be a simple operation (e.g., *button click*), or an operation with arguments (e.g., *text input*). Finally, if the action of a GUI event is an assertion, e.g., *isDisplayed*, we categorize the event as an *oracle event*.

To provide background knowledge about test transfer and illustrate the new challenges when the transfer is across platforms, consider WordPress [10], a popular blog and content management system. Figure 1 shows the excerpted steps to save a draft in WordPress using its web app. The user first gets to the post-listing page and then clicks the "Add New" button to initiate a new blog. After typing in the title and content, she clicks the "Save Draft" button and finally navigates back to the post-listing page to ensure the draft is saved. Figure 2 depicts how the same functionality is performed and tested on the Android app of WordPress. Table I shows the GUI and oracle events for this functionality on the two platforms.

As shown in Table I, while the core steps to perform this functionality on these two platforms are conceptually identical, automatically reusing the source test for web app to generate the target test for Android app is hindered by several challenges. A critical challenge that has been addressed by prior work [7], [8] is the mapping of syntactically different but semantically similar GUI widgets between apps, such as the "Add New" and "Create a Post" buttons in Table I. By leveraging advances in natural language processing (described in Section IV), prior work has shown the possibility of resolving such non-trivial mappings to transfer the GUI events.

Nevertheless, prior techniques have not addressed several challenges that are unique to cross-platform transfer. First,

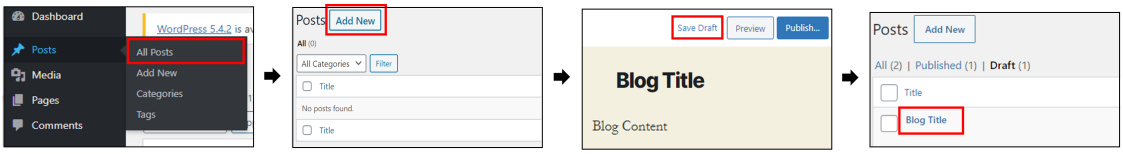


Fig. 1: Saving a draft with the web app of WordPress

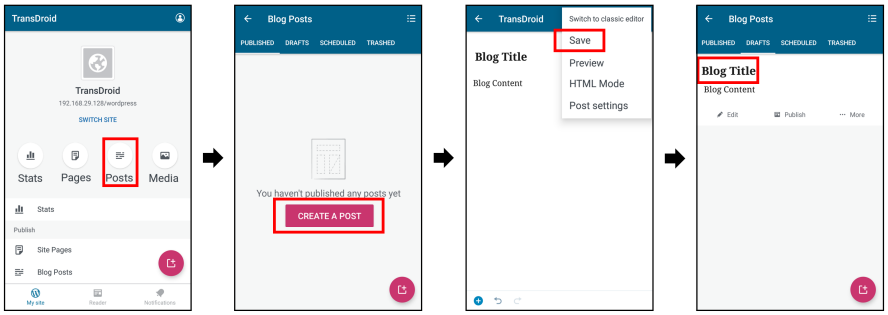


Fig. 2: Saving a draft with the Android app of WordPress

here the source test contains actions that do not exist on the Android platform, i.e., *mouseOver* and *jumpByURL*. Second, sometimes the contextual information about the source widgets are insufficient for guiding the search and mapping of the target widgets. For instance, if the source widget is a `<span>` HTML tag with text "Posts", and there are two target widgets, an `Android.widget.TextView` with text "Blog Posts" and an `Android.widget.Button` with text "Create a Post" (just as shown in the second screen of Figure 2), it is difficult to determine which one is the corresponding target widget. In other words, the behavior of the source tag (i.e., `<span>`) is unclear and other attributes, such as text values, are insufficient for determining the proper target widget.

We have designed TRANSDROID to overcome the aforementioned challenges and transfer such tests from web to Android. For example, the first *mouseOver* event and the second *click* event in the source test are merged, and then transferred as the first *click* event on the target app. Furthermore, the *jumpByURL* event in the source test is first converted to an intermediate event *navigateToActivity* on Android, and then transferred as the *click* event on the target app.

III. APPROACH

Figure 3 provides an overview of TRANSDROID. It takes a source test, a source app, and a target app as input, and generates a target test that examines the same functionality as the source test on the target app. TRANSDROID consists of four components: *Context Extraction*, *Action Transformation*, *NavGraph Extration*, and *Test Generation*. We describe the implementation of each component in the following subsections.

A. Context Extraction

Context Extraction component execute the source test and retrieves the contextual information related to each event in the source test. Like prior work [7], [8], we extract *widget context* that comes from the attributes of the GUI widgets

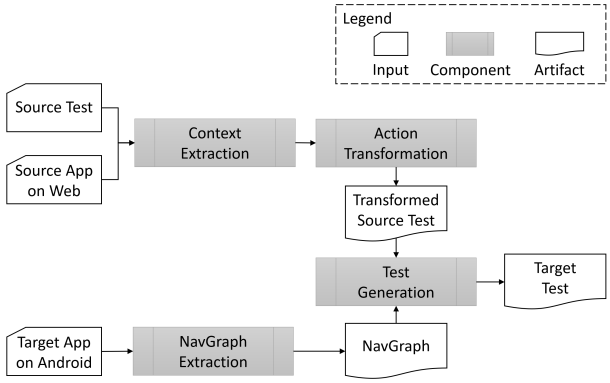


Fig. 3: Overview of TRANSDROID

interacted by the source test. However, unlike prior work, we additionally extract two other types of contextual information: *screen context* and *action context*. Screen context comes from the attributes of the GUI screens visited by the source test. Action context simply comes from the actions in the source test.

While we share the same insight as prior work that widget context can help identify correct target widget in test transfer, we believe that screen and action contexts, which are missing from the prior work, are also important to address the new challenges posed by cross-platform transfer. If we perceive the execution of a test as traversal through a graph consisting of an app's GUI screens, screen and action contexts provide additional information about how the path is visited. Following this insight, including the screen context in our analysis allows us to support source events that have no associated GUI widgets, such as the *jumpByURL* event mentioned previously. Furthermore, including action context in our analysis allows us to supplement our knowledge of the behaviorally ambiguous widgets. Taking the `<span>` tag described in Section II as an example, if its accompanied action is *click*, the search for

```
{
  "widget-context": {
    "class": "wp-menu-name",
    "text": "Posts"
  },
  "screen-context:": {
    "title": "Dashboard",
    "h1": "Dashboard"
  },
  "action-context": "mouseOver"
}
```


Fig. 4: Retrieved contexts of ("Posts", *mouseOver*) in Table I

the target widgets on Android can be limited to *clickables*. On the other hand, if the accompanied action is *input*, the search for the target widgets can be limited to *editables* such as `Android.widget.EditText`. Our transfer algorithm, thus, relies on all three forms of contextual information for search and mapping of the widgets.

In TRANSDROID, the Context Extraction component is implemented for web apps. It instruments and executes the source test to retrieve the contextual information related to each event. The widget context for GUI widgets in web apps, i.e., HTML elements, comes from their attributes, such as *id*, *name*, *class*, *href*, *placeholder*, etc., as well as the enclosed text. Moreover, the screen context in web apps are the title of the page (i.e., `<title>` tag) and first header element (e.g., `<h1>` tag), since they indicate the primary semantics of the screens. The action context in web apps are the actions performed by the source test, such as *click*, *input* and *mouseOver*. For example, Figure 4 shows the retrieved contexts of the first event in Table I, i.e., ("Posts", *mouseOver*).

B. Action Transformation

Action Transformation component processes the source test and ensures that the actions in the transformed source test are compatible with the target platform.

To better understand how the actions between web and Android should be transformed, we systematically inspected all input events supported by Selenium [17] (a widely-used framework for web automation testing) and Android [18]. Following our inspection, we identified four basic operations that can be used to define a set of customizable rules for the transformation: *reuse*, *merge*, *conversion*, and *removal*. First, the actions commonly shared by GUI-based software such as *click* can be directly reused on the target platform. Next, if an event contains a platform-specific action, it is possible to combine the event with its preceding or succeeding event (i.e., merge). Alternatively, we may replace the action with a similar action available on the target platform (i.e., conversion). Finally, if the incompatible action is not suitable for merge or conversion, the event may be removed from the transformed test.

Table II shows the action transformation rules adopted by TRANSDROID. An example of merge action is *mouseOver*, since this action is typically followed by *click* to form a common combo operation on web apps to open a sub-menu.

TABLE II: Action Transformation Rules in TRANSDROID

Source Action on Web	Target Action on Android	Operation Type
click()	*click()*	reuse
textInput()	*textInput()*	reuse
mouseOver()	(merge into the next source action)	merge
rightClick()	(merge into the next source action)	merge
jumpByURL()	*navigateToActivity()*	conversion
doubleClick()	*click()*	conversion
keyDown()	(removed)	removal
switchToWindow()	(removed)	removal

Therefore, a *mouseOver* event, together with the retrieved context, is merged with the following *click* event.

Examples of the converted actions include *jumpByURL* and *doubleClick*. Because the semantics behind *jumpByURL* is a change of GUI state, this action is converted to *navigateToActivity*, an intermediate action defined in TRANSDROID for Android with the similar intention. On the other hand, *doubleClick* in web apps is usually adopted to provide desktop-like user experience for features, such as opening a file in file manager or editing cells in a spreadsheet. Since such a user experience is rarely available in native mobile apps, we simply change *doubleClick* to *click*.

Finally, an instance of removed actions is *keyDown*, since it is usually used to perform a modifier key press (e.g., Shift) and may be safely removed without affecting the testing flow of the generated target test. Note that the presented action transformation rules can be extended for more platform-specific actions, or customized to accommodate the context of the target apps.

C. NavGraph Extraction

NavGraph Extraction component extracts the *Navigation Graph* of the target app. A Navigation Graph G of an app A is generally defined as a tuple (s, V, E) where:

- s denotes the starting state, i.e., the initial state after A has been fully loaded and started.
- V is a set of GUI states (screens). Each $v \in V$ represents a unique runtime GUI state in A. Moreover, each v is associated with a set of GUI widgets, W_v, that could be rendered in state v.
- E is a set of edges between the GUI states. Each $e = (v_1, v_2, (w, a)) \in E$ represents a transition from v_1 to v_2 by firing a GUI event (w, a).

We leveraged and modified the static analysis tool in prior work [8] to construct the Navigation Graph for the target app. The tool first extracts Activities in the target app as the GUI states. For each Activity, it then retrieves the associated GUI widgets (as well as their context if possible) from the resource files and source code. At last, it identifies transitions among Activities as the edges, by analyzing the registered event handlers on the widgets associated with each Activity. Figure 5 illustrates part of the Navigation Graph for WordPress on Android. This graph provides information about the screens and widgets comprising the target app to the Test Generation component, which as described next, applies a novel, heuristic-based algorithm to generate the target tests.

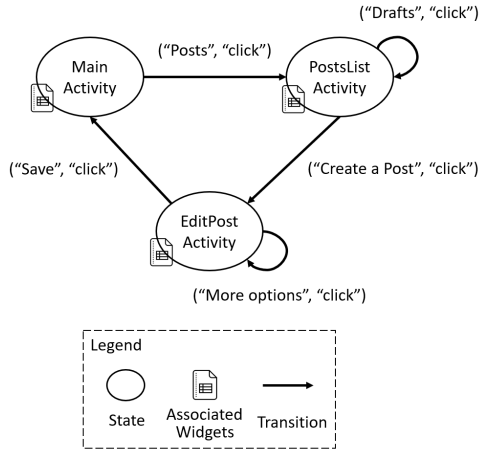


Fig. 5: Excerpted Navigation Graph for WordPress on Android

IV. Test Generation

Test Generation component takes a $targetApp$ and its corresponding $navGraph$, as well as a transformed source test t as input, and generates the target test t_n using a model-based, greedy search algorithm, as described in Algorithm 1. The algorithm consists of three main steps: (1) transfer the source events one-by-one to the target app; (2) update the Navigation Graph based on runtime information; and (3) repeat the transfer until no improvement can be made.

First, for each $event = (w_i, a_i) \in t$, if the event has an associated widget, i.e., w_i is not $null$, Algorithm 1 consults $navGraph$ with the widget context and action context of the event, and collects a list of widgets in the target app, $widgets$, in which the widgets are ranked based on their similarity to w_i (line 6-9). Next, for each $w_n \in widgets$, it checks whether w_n is reachable, and if so, identifies a sequence of events, $leadingEvents$, that should be executed to reach w_n (line 10-12). After that, it determines an appropriate action a_n for the identified w_n, and composes the $targetEvent = (w_n, a_n)$ (line 13-14), which will be added into t_n together with $leadingEvents$ (line 28). More details about similarity computation (lines 9 and 20), reachability check (lines 11 and 22), and action generation (line 13) are described in the next subsections.

On the other hand, if the source event has no widget attached, i.e., w_i is $null$, that means no target widget needs to be mapped. Instead, we consult $navGraph$ with the screen context of the event, and try to identify a reachable GUI screen s in the target app that is most similar to the screen visited by the source event (lines 18-26). In this case, $leadingEvents$ is a sequence of events that should be executed to reach s, (line 23) and $targetEvent$ is left to be $null$.

Once all the events in t are processed, the algorithm computes the fitness of the generated test t_n by evaluating its similarity to source test t, i.e., a weighted average of similarity scores (described in the next subsection) is computed for the corresponding events. If the fitness of t_n cannot be improved any further by a user-specified threshold or other termination criterion such as time limit is reached (line 30), the algorithm terminates and returns (line 34). Otherwise, it repeats the transfer and tries to find a better solution with an updated $navGraph$. As described in the following subsections, during the reachability analysis (lines 11 and 22), we also update $navGraph$ to improve the precision of our statically-retrieved app model with dynamically observed behaviors, thereby improving the likelihood of solving the search problem with each iteration of the algorithm.

Algorithm 1 Test Generation

Input:
$targetApp$, $navGraph$, and
Transformed source test $t = \{(w_1, a_1), (w_2, a_2), ...\}$
Output:
$t_n = \{(w_{n_1}, a_{n_1}), (w_{n_2}, a_{n_2}), ...\}$ for $targetApp$

```
1: while true do
2:     t_n = ∅
3:     for each event = (w_i, a_i) ∈ t do
4:         leadingEvents = ∅
5:         targetEvent = null
6:         if w_i is not null then
7:             x_w = getWidgetContext(event)
8:             x_a = getActionContext(event)
9:             widgets =
                   getSimWidgets(x_w, x_a, navGraph)
10:            for each w_n ∈ widgets do
11:                if isReachable(w_n, t_n, navGraph) then
12:                    leadingEvents =
                           getPath(w_n, navGraph)
13:                    a_n = generateAction(w_i, a_i, w_n)
14:                    targetEvent = (w_n, a_n)
15:                    break
16:                end if
17:            end for
18:        else
19:            x_s = getScreenContext(event)
20:            screens = getSimScreens(x_s, navGraph)
21:            for each s ∈ screens do
22:                if isReachable(s, t_n, navGraph) then
23:                    leadingEvents =
                           getPath(s, navGraph)
24:                    break
25:                end if
26:            end for
27:        end if
28:        t_n = t_n ∪ leadingEvents ∪ targetEvent
29:    end for
30:    if Δ fitness(t, t_n) ≤ threshold or timeout
31:        break
32:    end if
33: end while
34: return t_n
```

A. Similarity Computation

Similarity between GUI widgets or screens is primarily determined by their context. The similarity between two widgets is based on their widget context and action context. For two GUI screens, the similarity is determined by their screen context. Since the contexts are represented as words or word lists, the similarity can be computed by leveraging different metrics in natural language processing (NLP). In this paper, following the recent test transfer works [7], [8], we leverage Word2Vec [19] to compute the similarity between two

Algorithm 2 Function: isReachable()

```
1: function ISREACHABLE(entity, t_n, navGraph)
2:     execute(t_n)
3:     curScreen = getCurrentGUIScreen()
4:     if entity is a widget then
5:         dstScreen = getGUIScreen(entity, navGraph)
6:     else                                  ▷ entity is a GUI Screen
7:         dstScreen = entity
8:     end if
9:     paths = getPaths(curScreen, dstScreen, navGraph)
10:    for each path ∈ paths do
11:        isValid = validate(entity, path, navGraph)
12:        if isValid is true then
13:            return true
14:        end if
15:    end for
16:    return false
17: end function
```

widgets or GUI screens. Word2Vec is a neural network model that captures the linguistic contexts of words. In this model, each word is represented as a real-value vector (called word embedding). A characteristic of Word2Vec is that semantically related words are close together in terms of their cosine similarity. For instance, "Create" is closer to "Add" (with cosine similarity of 0.47) than "Delete" (with cosine similarity of 0.33) in the vector space. As a result, even if the "Add New" button does not exist in the Android App, TRANSDROID can still find its most similar widget, i.e., the "Create a Post" button, as shown in the motivating example (the second row in Table I).

To exemplify how we compute the similarity between two contexts, consider the "Add New" button and the "Create a Post" button in the motivating example (the second row in Table I). To compute the similarity between their text, i.e., "Add New" and "Create a Post", we first apply a series of common practices in NLP, including tokenization and stopword removal, to convert the text into word lists, i.e., ["Add", "New"] and ["Create", "Post"]. Next, we query a pre-trained Word2Vec model released by Google [20] to obtain the cosine similarity scores for the word pairs as follows:

$$\begin{array}{c} \\ Create \\ Post \end{array} \begin{array}{c} \begin{array}{cc} Add & New \end{array} \\ \begin{bmatrix} \mathbf{0.47} & 0.22 \\ 0.10 & \mathbf{0.12} \end{bmatrix} \end{array}$$

The similarity for the text is then calculated as $(0.47 + 0.12)/2 = 0.29$, the average of the pairs with the highest score. We compute the similarity scores for other attributes of these two buttons following the same way. The final similarity score is calculated as a weighted sum of the scores from all of their attributes.

B. Reachability Check

The Navigation Graph needs to be verified and updated during transfer, because the graph may be initially derived through static analysis, which tends to over-approximate the app's runtime behavior. Algorithm 2 describes the function `isReachable` called on lines 11 and 22 of Algorithm 1. This function serves two main purposes: (1) check if an $entity$, i.e., a widget or GUI screen, is reachable by verifying the possible paths leading to it; and (2) update $navGraph$, including the events that trigger transitions between screens as well as the associated widgets with each GUI screen, during the verification.

To that end, the function first restarts the app and executes t_n, i.e., the events successfully transferred so far, to get to the current GUI screen, $curScreen$ (line 2-3). Next, if $entity$ is a widget, the function assigns the GUI screen associated with $entity$ to the destination screen, $dstScreen$ (line 5); otherwise the $entity$ itself is assigned to $dstScreen$ (line 7). After that, all possible paths between $curScreen$ and $dstScreen$ are executed to verify whether $entity$ is reachable. The function returns $true$ once a feasible path for $entity$ is found; otherwise it returns $false$ (line 9-16). Moreover, the function `validate` in line 11 updates $navGraph$ by (1) removing unreachable paths; and (2) adding newly encountered widgets at runtime to the associated widgets of the GUI screen.

C. Action Generation

The function `generateAction` in line 13 of Algorithm 1 determines a proper action a_n for the identified target widget w_n. Typically, if the source event (w_i, a_i) contains a generic action such as *click* or *text input*, the action can be directly reused, i.e., $a_n = a_i$. However, it is possible that the correct action for a_n is other type of actions supported by or registered on w_n, such as $longClick$. On the other hand, a_n can be an assertion (e.g., *isDisplayed*) if the source event is an oracle event. Therefore, the implementation of `generateAction` needs to accommodate these situations.

In our implementation, for GUI events, we first analyze the source code[1] to check whether the identified target widget has specific event listeners, such as *setOnLongClickListener()*, and if so, we assign the action corresponding to such an event listener as the target action a_n. Otherwise, we reuse the source action.

If the source event (w_i, a_i) is an oracle event, i.e., a_i is an assertion, this function needs to be customized for different types of assertion. Currently, TRANSDROID supports four types of assertion commonly used in web testing [22], as shown in Table III. The first two assertion types, *isAttrEqual* and *isDisplayed*, are widget-specific, and their arguments needs to be modified when transferred to the target app. The other two assertion types, *textPresent* and *textNotPresent*, are widget-irrelevant assertions and can be directly transferred to the target app.

D. Walk-Through of the Motivating Example

We provide a walk-through of how TRANSDROID transfers the test shown in our illustrative example (recall Figures 1 and 2) to help the reader see the entire framework in action. First, Context Extraction executes the source test shown in Table I on the source app to retrieve the contextual information related to each event, and annotates the source test with

[1]Our analysis is actually performed on decompiled binary code (i.e., APKs) using Soot, a static analysis framework for Java [21].

TABLE III: Assertion types supported by TRANSDROID. (w_i, a_i): source oracle event. (w_n, a_n): transferred target event.

a_i	a_n
$isAttrEqual(VALUE_i, attr(w_i))$	$isAttrEqual(VALUE_n, attr(w_n))$
$isDisplayed(w_i)$	$isDisplayed(w_n)$
$textPresent(STRING)$	$textPresent(STRING)$
$textNotPresent(STRING)$	$textNotPresent(STRING)$

this information. After that, Action Transformation parses the source test and transforms it to an Android-compatible test, i.e., a test in which all of the actions are supported in Android. Particularly, the first *mouseOver* event is merged into the following *click* event, and the *jumpByURL* event is converted to *navigateToActivity* event. At the same time, NavGraph Extraction statically retrieves the Navigation Graph of the target app as the input for Test Generation.

In Test Generation, each event in the transformed source test is transferred one-by-one. The target widgets or GUI screens most similar to the source contexts are identified from the Navigation Graph with our formula for computing similarity. For example, when transferring the *navigateToActivity* event (transformed from *jumpByURL*), the algorithm searches for an Android Activity that is most similar to the screen context retrieved from `all-posts.php` (i.e., Activity with a name that is most similar to title/heading of php file), and generates the events leading to that screen. This results in a click on "Posts" button, which initiates a transition from *MainActivity* to *PostsListActivity*, as shown in Figure 5. All other events are similarly transferred. Finally, the last oracle event in the source test, i.e., existence check for the `<a>` tag with text *"Blog Title"*, is transferred to an existence check for the `android.widget.TextView` with the same text.

V. EVALUATION

We investigate the following research questions in our experimental evaluation of TRANSDROID:

RQ1. How effective is TRANSDROID in terms of (1) the precision and recall for widget mapping, and (2) the number of successful transfers compared to total attempted transfers?

RQ2. What are the main reasons behind transfer failure?

RQ3. How much effort can be saved by using TRANSDROID to generate tests?

RQ4. How efficient is TRANSDROID in terms of the running time to perform cross-platform transfer?

A. Experimental Setup

We implemented TRANSDROID with Python and Java for web tests written using Selenium [23]. Existing test cases for the subject apps are written with Selenium's Python client. The transferred Android tests are stored in JSON format and executed by our test runner implemented with Appium [24]. In our experiments, we used ChromeDriver [25] to execute the web tests, and a Pixel 2 Emulator running Android 7.1 (API 25) for test generation. The experiments were conducted on a Windows laptop with 2.8 GHz Intel Core i7 CPU and 32 GB RAM. Our experimental data is publicly available [16].

Subject apps and test suites. As noted by others [26], [27], it is very difficult to find publicly available web apps that have working UI test suites. We managed to find 10 pairs of web and Android apps (20 in total) with either existing tests, or existing documentation containing the test descriptions. Specifically, we reused the existing tests of DocuWiki [28], GitLab [29] and OwnCloud [30], and created the tests for other subject apps by following the test descriptions found in prior work [6], [31]. Table IV shows the 10 pairs of real-world subjects used in our study, including the size of test suites and the number of GUI and oracle events. There are 110 web tests in total, each containing 5.1 events (including 1.8 oracle events) on average. The features examined by the test suites can be found on TRANSDROID's website [16]. In addition, we manually constructed the corresponding Android tests. These tests served as the ground truth in our experiments to determine if (1) the target widgets were correctly identified, and (2) the transfers were successful or not (detailed in the next paragraph).

Effectiveness of attempted transfers. TRANSDROID transfers each of the tests for a web app to its Android counterpart, resulting in 110 total attempted transfers. For each transfer, we used the ground truth to examine the generated test to identify *false positive*, *false negative*, and *true positive* cases as follows: *false positive* occurs when the target widget of manual transfer is different from the widget identified by TRANSDROID; *false negative* occurs when TRANSDROID fails to find a target widget, while manual transfer can; and *true positive* occurs when the target widget from manual transfer matches the widget identified by TRANSDROID. Based on these metrics, we measured the *Precision* as the number of generated target events that are correct. Additionally, *Recall* measures how many of the source events are correctly transferred.

Precision and recall can faithfully evaluate the correctness of widget mapping, but not necessarily the successfulness of test transfer [8], [31]. In other words, precision and recall do not consider whether the generated tests are executable or applicable in the context of the target app. For example, suppose a web app requires the user to provide a password only once during registration, but twice on its Android counterpart for confirmation. In that case, the transfer may have very high precision and recall if most of the source events are correctly migrated. However, the generated test on Android is still not executable because it lacks the required password confirmation step. As a result, we also report whether the attempted transfers were successful by manually examining the generated tests. A successful transfer means that the generated test was executable and actually meaningful in the context of the target app, verifying the same feature as the source test.

Effort Reduction. Another perspective to evaluate the usefulness of TRANSDROID in practice is to measure how much effort developers can save if they adopt this tool to generate tests instead of writing them from scratch, regardless of whether the transfers are successful or not. To that end, we first measure how close a transferred test is to its ground-

TABLE IV: Subject apps and test suites

Subject App	Description	Web Version	Android Version	#Web Tests	#Events in Web Test		
					GUI	Oracle	Total
BuzzFeed	News and entertainment	Live website	com.buzzfeed.android:v2021.3	11	30	19	49
DokuWiki	Collaborative editor	v2018-04-22	com.fabienli.dokuwiki:v0.10	9	29	17	46
Etsy	E-commerce	Live website	com.etsy.android:v5.53.1	13	40	18	58
Fox News	News television channel	Live website	com.foxnews.android:v4.22.0	11	30	17	47
GitLab	DevOps lifecycle tool	v13.2.2	com.commit451.gitlab:v2.6.3	11	38	27	65
Groupon	E-commerce	Live website	com.groupon:v20.10.224420	13	39	23	62
Hacker News	Social news forum	Live website	net.dreambits.hackernews:v2.5	12	39	24	63
OwnCloud	File hosting	v10.5	com.owncloud.android:v2.15	11	40	20	60
Wikipedia	Online encyclopedia	Live website	org.wikipedia:v2.7.50320	9	37	14	51
WordPress	Content management	v5.3.2	org.wordpress.android:v14.3	10	43	17	60
Total				110	365	196	561

TABLE V: Effectiveness evaluation of TRANSDROID

Subject	GUI Event		Oracle Event		#Successful Transfer
	Precision	Recall	Precision	Recall	
BuzzFeed	64%	95%	63%	100%	64% (7/11)
DokuWiki	70%	90%	94%	94%	89% (8/9)
Etsy	95%	100%	94%	100%	100% (13/13)
Fox News	86%	100%	71%	100%	64% (7/11)
GitLab	81%	100%	81%	100%	64% (7/11)
Groupon	74%	100%	87%	100%	69% (9/13)
Hacker News	79%	100%	83%	100%	67% (8/12)
OwnCloud	71%	96%	85%	100%	73% (8/11)
Wikipedia	86%	100%	92%	92%	89% (8/9)
WordPress	88%	100%	100%	100%	100% (10/10)
Total	80%	99%	85%	99%	77% (85/110)

truth test by computing their Levenshtein distance [32] or edit distance. Levenshtein distance is a string metric to compute the minimum number of edits required to change one word to another word. In our case, a single edit is defined as an insertion, deletion or substitution of an event in the transferred test. Next, we further define *reduction of effort* as follows:

$$Reduction(t_n) = 1 - \frac{editDistance(t_n, t_g)}{\#events(t_g)}$$

This equation measures the manual effort reduced by a generated test t_n compared to writing its ground truth t_g from scratch. For example, if a 6-event generated test needs 2 edits (e.g., 1 deletion and 1 substitution) to its 5-event ground truth, compared to writing the ground truth from scratch, the reduced manual effort through the generated test is $1 - (2/5) = 60\%$.

B. RQ1: Effectiveness

Table V demonstrates the effectiveness of TRANSDROID in terms of precision, recall, and successful transfers for each subject listed in Table IV. These results show that in total, 77% of the attempted transfers by TRANSDROID are successful, with an overall 82% precision and 99% recall considering all the transferred GUI and oracle events. TRANSDROID is substantially effective in identifying correct GUI widgets and successfully transferring tests from web to Android.

Interestingly, we found that perfectly matching all the widgets and screens in a source test is not always necessary to successfully transfer the test. Two instances for such cases are WordPress and Etsy, in which all the tests were successfully transferred (i.e., 100% success rate), despite the existence of several false positives in the matched GUI events (i.e., imperfect precision). The reason is that sometimes false positives are not harmful, since the same feature may be implemented differently on two platforms. For example, to access the "About Me" page on WordPress's web app, users need to expand and navigate the side menu, which is not required on the Android app, as it provides a shortcut to that page on its main screen.

Another important observation from the results in Table V is that the success rate varies among different subjects, ranging from 64% (on BuzzFeed, Fox News, and GitLab) to 100% (on Etsy and WordPress). In the next research question, we investigate the attributes that impact the success rate of test transfer.

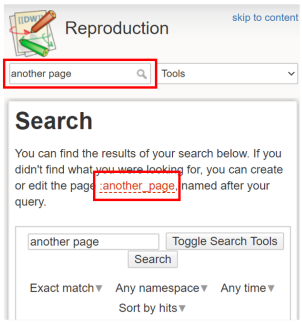
C. RQ2: Factors Impacting Effectiveness

We manually investigated all of the attempted transfers, including both successful and failed ones, to identify the factors that impact the effectiveness.

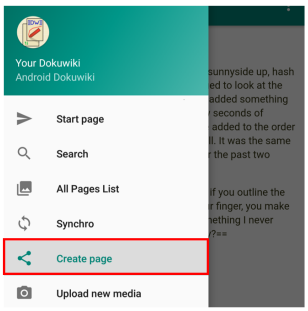
Insufficient widget context. Insufficient contextual information in the target widgets, such as indistinguishable or missing textual information, impacts TRANSDROID's ability to find a proper match. For instance, while the input fields for shipping address in Groupon's web app contain distinguishable identifiers such as `city` and `zip-code`, the Android app uses the same identifier, i.e., `edit-text`, for all of the corresponding fields. As a result, TRANSDROID failed to transfer the tests dealing with the shipping feature. Another example is the Navigation Drawer (a.k.a., the menu icon or hamburger icon) in GitLab's Android app. It is implemented as an image button, rather than a native Android icon, without any associated textual information. As TRANSDROID only considers textual information for widget context, this button could not be matched and the features accessible through this button remain undiscovered.

Missing features. The test transfer fails if the tested feature is not implemented in the target app/platform. For example, the web users of OwnCloud can restore deleted files from the recycle bin, but this feature is not provided on OwnCloud's Android app. Another instance of such transfer failure is the "unvote" feature (i.e., to revoke the vote for a news post) that is only provided on the web app of Hacker News, and not its Android counterpart.

Radically different interaction flows. If the interaction flow of accessing a feature is utterly different across platforms, the transfer may fail. For example, to create a new page on DokuWiki's web app, users need to first search for the



(a) Create a page on the web app through the search bar and a dynamically generated link



(b) Create a page on the Android app through a button

Fig. 6: Different interaction flows to create a page on Dokuwiki across platforms

TABLE VI: Average reduction of effort

Subject	#Events on Average			%Reduction
	Generated Test	Ground Truth Test	Edit Distance	
BuzzFeed	5.13	4.13	1.75	58%
DokuWiki	6.11	5.89	0.78	87%
Etsy	4.62	4.46	0.23	95%
Fox News	4.89	4.67	0.67	86%
GitLab	6.27	6.09	1.27	79%
Groupon	5.23	4.85	1.08	78%
Hacker News	6.82	6.55	1.00	85%
OwnCloud	6.80	6.30	1.50	76%
Wikipedia	6.33	5.89	0.89	85%
WordPress	6.50	5.60	0.90	84%
Total	5.84	5.44	0.98	82%

name of the page that they want to create, and then click the link dynamically generated in the search result. This form of interaction, however, is not supported by the Android app. A new page can only be created by clicking the "Create page" button on the Android app, as shown in Figure 6.

Test length is NOT a key factor. Prior work in intra-platform test transfer [8] found a strong negative correlation between test length (i.e., number of total events) and the effectiveness metrics (i.e., precision, recall, and success rate). Their finding inspired us to investigate if a similar correlation can be found in inter-platform test transfer. We conducted a Pearson correlation analysis [33] on our dataset, and observed a negligible correlation with the coefficients ranging between 0.03 and 0.30. Since it appears test length is not a key factor impacting effectiveness of cross-platform test transfer, we argue that future research should focus on other factors to improve the success rate of cross-platform transfer.

D. RQ3: Reduced Effort

Table VI demonstrates the average number of events comprising the ground-truth tests and transferred tests, along with their edit distance. The results show that TRANSDROID can save 82% of the manual effort on average, compared to writing the ground-truth tests from scratch. Taking WordPress as an example, on average the tests generated by TRANSDROID contain 6.5 events and need only 0.9 manual edits to be transformed to the ground truth, thereby achieving a substantial reduction in the manual effort of creating the tests from scratch. This result hints at the potential utility of TRANSDROID, even when it fails to successfully transfer the entire test suite.

TABLE VII: Efficiency evaluation of TRANSDROID

Subject	Time in Sec for a Transfer		
	Min	Max	Avg
BuzzFeed	48	3160	518
DokuWiki	14	2060	396
Etsy	17	271	95
Fox News	50	1724	639
GitLab	13	157	66
Groupon	54	644	199
Hacker News	21	424	133
OwnCloud	42	3132	629
Wikipedia	51	4445	769
WordPress	18	613	212
Total	13	4445	349

E. RQ4: Efficiency

Table VII shows the execution time for the attempted transfers in our experiments. On average, a test transfer takes less than 6 minutes, ranging from 13 seconds to 1.2 hours among the different tests and subject apps. Note that, however, among all 110 attempted transfers, only 4 of them took more than half an hour (the four largest numbers shown in Table VII). The average execution time for the other 106 transfers is only 241 seconds or approximately 4 minutes. While it is not feasible to directly compare our inter-platform transfer work against prior works targeting intra-platform transfer, the efficiency demonstrated by TRANSDROID is quite impressive, since prior work for intra-platform transfer takes 1.5 hours on average [8] to finish a transfer.

We investigated the four longest transfers and found that they spent most time in checking the reachability of many candidate widgets. That is, they ran the function `isReachable` in Algorithm 2 repeatedly. This is the most time-consuming element of the Test Generation component in general, as it frequently restarts the target app to validate the potential paths for a candidate widget or screen. Nevertheless, such reachability checks may be accelerated if executed in parallel with multiple devices or emulators.

VI. THREATS TO VALIDITY

The major external threat to validity of our results is the generalization to other subject apps and test cases. To mitigate this threat, we collected both commercial and open-source subjects under various categories. We also reused existing tests and created tests by closely following the documented specifications. Moreover, TRANSDROID also assumes the interaction flows between the web and the Android versions of an app are similar, albeit not identical, for a given feature. As discussed in RQ2 (Section V-C), we acknowledge that the same feature may be realized with drastically different interaction flows on different platforms. Nevertheless, as our evaluation shows, that is typically not the case in practice, and the proposed transfers are effective on a considerable number of apps that have similar cross-platform behaviors. The main internal threat

to validity of the proposed approach is the possible mistakes involved in our implementation and experiments. We manually inspected all of our results to increase our confidence in their correctness. The experimental data is also publicly available for external inspection.

Similar to all prior work in intra-platform test transfer [3]–[8], TRANSDROID assumes the source web app and its Android counterpart have similar features. If not, test transfer would not work. In some cases the Android app may have unique and platform-specific features, such as notification- or geolocation-related functionalities, for which test transfer will not be possible, since the corresponding web app lacks those features and hence does not have any related tests for transfer. Nevertheless, the goal of this work is to reduce the manual effort of writing tests for features that are shared.

The current implementation of TRANSDROID does not support some actions in web testing such as drag and drop. Furthermore, this paper does not consider external communications in the source web test, such as choosing a local file or login with OAuth. Such limitations could be addressed by extending the current action transformation rules. That said, one can trivially construct an automated pre-processing phase to exclude the source tests containing unsupported actions to improve the success rate of transfers.

VII. RELATED WORK

Intra-platform test transfer. In recent years, researchers have proposed approaches for transferring or reusing GUI tests for apps within a platform [3]–[8], [31]. Rau et al. [3] proposed a technique for mapping of GUI widgets among web applications. Hu et al. [5] presented a framework that leverages machine learning to synthesize reusable UI tests for Android apps. Qin et al. [6] proposed TestMig to migrate GUI events for the different instances of the same app running on iOS and Android. Behrang and Orso [4] proposed an approach to migrate test cases by mapping the GUI widgets to support assessment of mobile app coding assignments. That work was later extended to general test transfer between similar Android apps [7], [8]. A recent work by Zhao et al. [31] presented a framework as well as dataset to evaluate several test transfer techniques for Android apps. Unlike all prior work, TRANSDROID targets the needs and new challenges for cross-platform, i.e., web-to-Android, test transfer.

The key novelty of this paper is that we consider and address the new challenges in cross-platform transfer, i.e., unclear widget context and incompatible actions. First, unlike previous work such as [7], we propose to include two new types of contextual information (screen context and action context) in test generation. Moreover, we introduce Action Transformation, a pre-transfer phase with customizable rules to accommodate incompatible actions between different platforms. Note that, while at a high level we adopt the techniques similar to [7], e.g., combination of static and dynamic analyses, TRANSDROID is different from [7] in terms of several algorithmic details. For example, in [7], the statically extracted model of app is used as secondary information only when the dynamic exploration fails to find a match of widget. In contrast, TRANSDROID leverages the result of static analysis as primary information to construct Navigation Graph, which is the basis of the proposed algorithm.

Cross-platform testing. A significant number of previous studies have focused on cross-platform testing [34]–[40]. Developed concurrently, CrossT [34] and WebDiff [35] addressed the problem of cross-browser inconsistencies (XBIs) in web apps, i.e., the same web app behaving differently on different browsers. CrossCheck [36] combined the approaches in CrossT and WebDiff, and leveraged machine learning techniques such as decision tree to improve the accuracy of the reported XBIs. Later, based on an extensive study of XBIs in real-world web apps, X-PERT [37] proposed a framework applying different techniques for different types of XBIs to increase the effectiveness. Regarding the similar presentation issues on Android apps caused by device fragmentation, DiffDroid [38] combined input generation and differential testing to find cross-device inconsistencies. On the other hand, FMAP [39] analyzed the client-server communication of the desktop version and mobile version of a web app to identify missing features in either version. Similar to FMAP, CheckCAMP [40] proposed to identify missing functionality in either the iOS or Android version of the same app. However, none of these works aim to reuse and migrate existing tests across platforms.

VIII. CONCLUSION

Automated test transfer is a promising method of generating high-quality tests for verification of similar features among mobile apps. In this paper, we described the cross-platform test transfer problem and the associated challenges that prior works have not addressed. We presented TRANSDROID, an automated tool for solving this problem in the context of web-to-Android transfer. TRANSDROID adopts novel transformation techniques and test generation algorithms to resolve the challenges of overcoming the incompatibilities between platforms. Our evaluation on real-world apps demonstrated the effectiveness and efficiency of TRANSDROID, as it successfully transferred 77% of the test cases in our experiments. Our results indicate that even when test transfer is not completely successful, it has the potential of significantly reducing the manual effort of creating tests for similar apps, i.e., 82% reduction on average in our study subjects.

We also discussed the factors impacting the effectiveness of TRANSDROID, addressing which will frame part of our future work. We also aim to conduct a user study involving real developers to further validate our empirical findings. Finally, we plan to investigate the application of TRANSDROID for test transfer among other platforms, such as mobile-to-web and web-to-desktop.

ACKNOWLEDGMENT

This work was supported in part by award number CCF-2106306 from the National Science Foundation. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES

[1] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 613–622.

[2] J. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source android apps: A large-scale empirical study," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[3] A. Rau, J. Hotzkow, and A. Zeller, "Transferring tests across web applications," in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds. Cham: Springer International Publishing, 2018, pp. 50–64.

[4] F. Behrang and A. Orso, "Test migration for efficient large-scale assessment of mobile app coding assignments," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 164–175. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213854

[5] G. Hu, L. Zhu, and J. Yang, "Appflow: Using machine learning to synthesize robust, reusable ui tests," in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ser. ESEC/FSE 2018, 2018.

[6] X. Qin, H. Zhong, and X. Wang, "Testmig: Migrating gui test cases from ios to android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 284–295. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330575

[7] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 54–65.

[8] J. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 42–53.

[9] (2020) Alexa top sites in the us. [Online]. Available: https://www.alexa.com/topsites/countries/US

[10] (2020) Wordpress.org. [Online]. Available: https://wordpress.org/

[11] B. Fling. (2009) A brief history of mobile. [Online]. Available: https://www.oreilly.com/library/view/mobile-design-and/9780596806231/ch01.html

[12] D. Hillis. (2011) Mobile internet era: Planning your mobile strategy. [Online]. Available: https://www.cmswire.com/cms/web-engagement/mobile-internet-era-planning-your-mobile-strategy-010147.php

[13] (2020) Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia

[14] (2020) Twitter. [Online]. Available: https://twitter.com

[15] (2020) Zoom. [Online]. Available: https://zoom.us/

[16] https://sites.google.com/view/icst22-transdroid, 2021.

[17] (2021) Keyboard and mouse actions api in selenium. [Online]. Available: https://www.selenium.dev/documentation/webdriver/actions_api/

[18] (2021) Input events overview. [Online]. Available: https://developer.android.com/guide/topics/ui/ui-events

[19] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119. [Online]. Available: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[20] (2020) Word2vec. [Online]. Available: https://code.google.com/archive/p/word2vec/

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[22] (2020) Common assertions. [Online]. Available: https://www.soapui.org/docs/functional-testing/validating-messages/getting-started-with-assertions.html/#3-Common-Assertions

[23] (2020) Selenium. [Online]. Available: https://www.selenium.dev/

[24] (2020) Appium. [Online]. Available: https://github.com/appium/appium

[25] (2020) Chrome web driver. [Online]. Available: https://chromedriver.chromium.org/home

[26] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'14. ACM, 2014, p. 67–78. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642991

[27] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 83–93. [Online]. Available: https://doi.org/10.1145/2771783.2771786

[28] (2021) Docuwiki tests. [Online]. Available: https://github.com/splitbrain/dokuwiki/blob/master/_test/tests/test

[29] (2021) Gitlab test spec. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/tree/master/qa/qa/specs/features

[30] (2021) Owncloud tests. [Online]. Available: https://github.com/owncloud/core/tree/master/tests/acceptance/features

[31] Y. Zhao, J. Chen, A. Sejfia, M. Schmitt Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic, "Fruiter: a framework for evaluating ui test reuse," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1190–1201.

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

[33] (2020) Pearson correlation coefficient. [Online]. Available: https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

[34] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. Association for Computing Machinery, May 2011, p. 561–570. [Online]. Available: https://doi.org/10.1145/1985793.1985870

[35] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *2010 IEEE International Conference on Software Maintenance (ICSM)*, Sep 2010, p. 1–10.

[36] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, Apr 2012, p. 171–180.

[37] S. R. Choudhary, M. R. Prasad, and A. Orso, "X-pert: Accurate identification of cross-browser issues in web applications," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 702–711.

[38] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, Oct 2017, p. 308–318.

[39] S. Roy Choudhary, M. R. Prasad, and A. Orso, "Cross-platform feature matching for web applications," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. Association for Computing Machinery, Jul 2014, p. 82–92. [Online]. Available: https://doi.org/10.1145/2610384.2610409

[40] M. E. Joorabchi, M. Ali, and A. Mesbah, "Detecting inconsistencies in multi-platform mobile apps," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, p. 450–460.