An Evolutionary Approach to Adapt Tests Across Mobile Apps

Leonardo Mariani*, Mauro Pezzè†‡, Valerio Terragni† and Daniele Zuddas†
*University of Milano Bicocca, Milan, Italy
†Università della Svizzera italiana, Lugano, Switzerland
‡Schaffhausen Institute of Technology, Schaffhausen, Switzerland
leonardo.mariani@unimib.it - {mauro.pezze, valerio.terragni, daniele.zuddas}@usi.ch

***Abstract*—Automatic generators of GUI tests often fail to generate semantically relevant test cases, and thus miss important test scenarios. To address this issue, test adaptation techniques can be used to automatically generate semantically meaningful GUI tests from test cases of applications with similar functionalities.**

In this paper, we present ADAPTDROID, a technique that approaches the test adaptation problem as a search-problem, and uses evolutionary testing to adapt GUI tests (including oracles) across similar Android apps. In our evaluation with 32 popular Android apps, ADAPTDROID successfully adapted semantically relevant test cases in 11 out of 20 cross-app adaptation scenarios.

***Index Terms*—GUI testing, test reuse, search-based testing, test and oracle generation, Android applications**

I. INTRODUCTION

Verifying GUI applications is both important, due to their pervasiveness, and challenging, due to the huge size of their execution space [1]. GUI testing is a popular way to verify the behavior of GUI applications, which amounts to design and execute GUI test cases. A GUI test case (GUI test in short) consists of (i) a sequence of events that interact with the GUI, and (ii) assertion oracles that predicate on the GUI state.

Because manually designing GUI tests is expensive, many automatic GUI test generators have been proposed. Current approaches generate GUI tests either randomly [2] or by relying on structural information that they obtain either from the GUI [3]–[5] or from the source code [6]. Current approaches suffer from two main limitations. By largely ignoring the semantics of the application, they produce tests that are either semantically meaningless or unrepresentative of the canonical usage of the application [7]. Thus, they likely miss the GUI event sequences that properly exercise functionalities and reveal faults [8], [9]. Moreover, current GUI test generators rely on implicit oracles [10], [11] that miss many failures related to semantic issues [9], [12].

Recently, researchers investigated the opportunity to address these challenges by exploiting semantic similarities across GUI applications [13]–[15]. Indeed, browsing the Google Play Store reveals many Android apps that are semantically similar, albeit offering different graphics appearance, access permissions, side features, and user experience [16]–[18]. Hu et al. have shown that among the top 309 non-game mobile apps in the Google Play Store, 196 (63.4%) of them fall into 15 groups that share many common functionalities [15]. This confirms the huge potential of sharing tests across similar applications because common functionalities yield to common GUI tests [15].

The recent CRAFTDROID [19] and APPTESTMIGRATOR [20] approaches generate GUI tests by automatically adapting existing GUI tests across similar Android apps. Both approaches generate new tests for a *recipient app*, by adapting the tests designed for some *donor app* that shares semantically similar functionalities with the *recipient app*. Test adaptation, when successful, addresses the limitations of existing GUI test generators: (i) it yields to semantically meaningful GUI tests that characterize canonical usages of the app under test. (ii) it leverages the functional oracles of the donor tests.

CRAFTDROID and APPTESTMIGRATOR explore a GUI model of the recipient app to find a sequence of events that maximize the semantic similarity with the events of the donor test. They compute the semantic similarity of GUI events using word embedding [21] applied to the textual descriptors of events extracted from the GUI widgets. Both techniques *greedily* explore a single test adaptation scenario, missing the many alternative adapted tests that could be generated starting from a same donor test. Indeed, extensively exploring the execution space is often imperative to identify a sequence of events that well reflects the semantics of the donor test.

In this paper, we present ADAPTDROID, a technique that formulates the GUI test adaptation problem as a search-problem using an evolutionary approach. ADAPTDROID explores the huge space of GUI tests with a fitness function that rewards the tests that are most similar to the donor test. The ADAPTDROID notion of similarity considers both the semantics of the events and the capability of the adapted test to reach states where the donor oracle can be applied to.

We implemented ADAPTDROID in a prototype tool, and evaluated with a human study involving 32 Android apps. Our results show that ADAPTDROID successfully adapts semantically relevant GUI tests in 11 out of 20 test adaptation scenarios. Thus confirming that test adaptation is a promising and complementary solution for generating GUI tests.

In summary, the main contributions of this paper are:

- formulating the problem of adapting GUI tests across similar applications as an evolutionary approach,
- proposing ADAPTDROID, to adapt both GUI event sequences and oracle assertions across mobile apps,

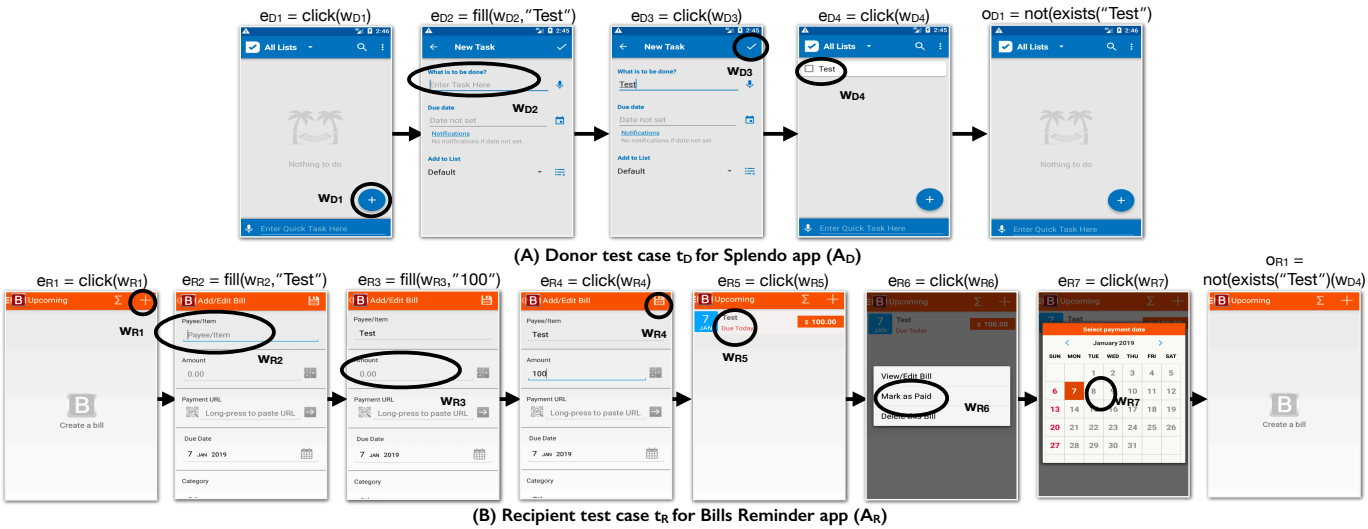

Fig. 1. Example of ADAPTDROID cross-application test adaptation

- presenting the results of a study showing that test adaptation of Android apps is a valuable opportunity,
- presenting an empirical evaluation of ADAPTDROID that highlights its effectiveness and limitations,
- releasing the ADAPTDROID tool and all experimental data [22].

II. ADAPTING TESTS ACROSS GUI APPLICATIONS

GUI applications interact with users through a Graphical User Interface (GUI) [23]. A **GUI** is a forest of hierarchical windows (*activities* in Android), where only one window is active at any time [3]. Windows host **widgets**, which are atomic GUI elements characterised by properties: *type*, displayed *text* (if any) and *xpath* (a label that uniquely identifies the widget in the structural hierarchy of the window [24]). At any time, the active window has a **state S** that encompasses the state (properties values) of the displayed widgets. Some widgets expose user-actionable events that users can trigger to interact with the GUI. For instance, users can click on widgets of type button or can fill widgets of type text field.

A **GUI test t** is an ordered sequence of events $\langle e_1, ..., e_n \rangle$ on widgets of the active windows. A test execution induces a sequence of *observable* state transitions $S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \ldots \xrightarrow{e_n} S_n$, where S_{i-1} and S_i denote the states of the active window before and after the execution of event e_i, respectively. An event is an atomic interaction on a widget. Events are typed. In this paper, we consider two common types of events

- ***click***(w): clicking a widget w;
- ***fill***$(w,\ txt)$: filling a string *txt* in widget w.

Each test t is associated with one or more **assertion oracles** [25] that check the correctness of the state S_n obtained after the execution of t [26]. We use O_t to denote the assertions associated with the test t, and consider two types of assertions:

- ***exists***(txt) checks if S_n contains a widget with text *txt*: $exists(txt) \Leftrightarrow \exists w \in S_n : text(w) = txt$;
- ***hasText***$(w,\ txt)$ checks if S_n has a widget w with text *txt*: $hasText(w,\ txt) \Leftrightarrow \exists w' \in S_n : xpath(w') = xpath(w) \wedge text(w') = txt$.

This paper presents ADAPTDROID to adapt GUI tests (including oracles) across interactive applications that implement similar functionalities. Given two Android apps A_D (donor), A_R (recipient), and a "donor" test t_D for A_D, ADAPTDROID generates a "recipient" test t_R that tests A_R as t_D tests A_D.

III. WORKING EXAMPLE

Figure 1 introduces a working example that illustrates the challenges of adapting GUI tests across similar applications. Figure 1A shows a donor GUI test (t_D) designed for *Splendo*, an Android app to manage tasks lists. The test *adds a new task to a task list, and verifies that the task disappears once marked as done*. Figure 1B shows how ADAPTDROID successfully adapts t_D to the recipient app *Bills Reminder* (A_R), by generating t_R that *adds a new bill to the bill list and verifies that the bill disappears once marked as paid*. Although the two apps belong to different domains, they share the logical operations of creating a new element (a task in A_D, a bill in A_R) and marking it as completed (done in A_D, paid in A_R). Automatically adapting GUI tests across apps presents three main challenges:

1) Huge space of GUI tests The space of the possible GUI tests grows exponentially with both the length of the donor test and the number of widgets in the recipient app [1]. Adapting tests requires an effective search strategy that recognizes the relevant GUI events in the recipient app.

2) GUI differences The donor test may exercise GUI widgets that are logically equivalent but very different from the widgets of the recipient app. For instance in Figure 1, semantically similar widgets are labelled "What is to be done?" (w_{D2}) and "Payee/Item" (w_{R2}), respectively. Also, *Splendo* uses a tick mark button (w_{D3}) to save a task, while *Bills Reminder* uses a floppy disk image button (w_{R4}).

3) No one-to-one GUI event matching The donor and adapted tests might have a different number of events. For instance,

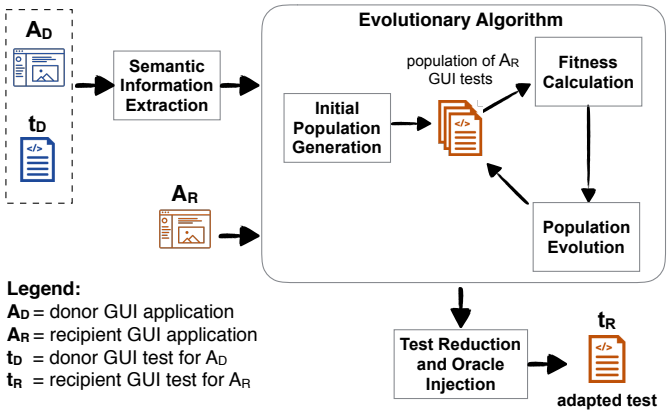

Fig. 2. The ADAPTDROID process

in Figure 1 the donor and recipient tests have four and seven events, respectively. Creating a bill in *Bills Reminder* requires more events than creating a task in *Splendo*. Marking a bill as paid in *Bills Reminder* requires a date, while marking a task as done in *Splendo* does not.

The next Section presents ADAPTDROID and discusses how it addresses these challenges.

IV. ADAPTDROID

Figure 2 overviews the ADAPTDROID process. ADAPTDROID takes as an input the APK of a donor application A_D, a donor test t_D and the APK of a recipient application A_R, and generates a test t_R for A_R.

ADAPTDROID adapts tests in five phases. The *Semantic Information Extraction* phase executes t_D on A_D to extract semantic information relevant to the adaptation process, such as the semantic descriptors of the widgets exercised by t_D. The *Initial Population Generation*, *Fitness Calculation* and *Population Evolution* phases implement an evolutionary algorithm that evolves a population of GUI tests guided by a fitness function that steers the evolution towards a test t_R as similar as possible to t_D. The *Population Evolution* and *Fitness Calculation* phases iterate until they either perfectly adapt the test (fitness = 1.0) or a time-budget expires. The *Test Reduction and Oracle Injection* phase removes irrelevant events in t_R and adds to t_R the oracles adapted from t_D.

ADAPTDROID faces Challenge 1 (huge space of GUI tests) with an *evolutionary algorithm equipped with a proper set of evolution operators*; Challenge 2 (GUI differences) with a *matching strategy that takes into account the semantics of GUI events*; Challenge 3 (no one-to-one GUI event matching) with a *flexible fitness function* that captures the different nature of the donor and recipient apps. The following subsections describe the cross-app semantic matching of GUI events and the five ADAPTDROID phases.

A. Cross-app Semantic Matching of GUI Events

ADAPTDROID matches GUI events across applications according to their *semantic similarity*, regardless of syntactic differences, such as, widget types, positions and layouts. This is because two similar apps may implement operations that are semantically similar but syntactically different.

ADAPTDROID encodes the semantics of an event as an ordered sequence of one or more words, a sentence in natural language, that we call *descriptor*. ADAPTDROID extracts the event descriptors from either the information shown in the GUI or the identifiers defined by the programmers (widget ids and file names). Given an event e_i, ADAPTDROID extracts its **descriptor $\mathbf{d_i}$** as follows.

For click events $e_i = click(w_i)$, d_i is the text displayed in the widget w_i (its text property). In the example of Figure 1, the label of widget w_{D4} "Test" is the descriptor of e_{D4}. Similarly, d_{R5} = "Test", d_{R6} = "Mark as Paid" and d_{R7} = "8". If the text property is empty and the widget w_i includes an image, d_i is the file name of the image. In the example of Figure 1, the name of the image file associated with w_{D1} ("bs_add_task") is the descriptor of e_{D1}. Similarly, d_{D3} = "action_save_task", d_{R1} = "action_add" and d_{R4} = "action_save". To facilitate the matching of descriptors, ADAPTDROID splits words by underscore or camel-case (d_{D1} becomes "bs add task"), removes stop-words, and performs lemmatization [27]. If the text property is empty and w_i does not include an image, d_i is the id assigned by the developers to reference w_i in the GUI.

For fill events $e_i = fill(w_i, txt_i)$, d_i is the text of the nearest widget from w_i. We follow the approach of Becce et al. [28], which is based on the observation that text fields are normally described by near labels. In the example of Figure 1, the text property of the label on top of w_{R3} "Amount" is the descriptor of e_{R3}. Similarly, d_{D2} ="What is to be done?"and d_{R2} = "Payee/Item". If there are no lables near w_i, d_i is the id assigned by the developers to reference w_i in the GUI.

ADAPTDROID identifies pairs of descriptors that represent the same concept with a Boolean function **ISSEMSIM*(txt$_1$,txt$_2$)*** that returns true if the sentences txt_1 and txt_2 are **semantically similar**, false otherwise. The many available distances, such as Euclidean Distance, Cosine Distance and Jaccard Similarity, are ill-suited for our purposes. This is because they are not designed to overcome the *synonym problem*, that is, two sentences have the same semantics but no common words [21]. We cannot expect that two distinct albeit similar apps use exactly the same words to express the same concepts.

Both CRAFTDROID [19] and APPTESTMIGRATOR [20] address the synonym problem with WORD2VEC [21], a vector-based *word embedding* [29]. WORD2VEC trains a model that embeds words into a vector space, where words with similar semantics are close in the space [21]. WORD2VEC matches single words, and thus it is inadequate when descriptors contain multiple words (as d_{R6} = "Mark as Paid" in Figure 1).

Instead, ADAPTDROID uses Word Mover's Distance (WMD) [30], which calculates the distance between sentences composed of one or more words [29]. Given two sentences txt_1 and txt_2, WMD returns a number between 0 to 1 that expresses how close the sentences are in the vector space [30]. ISSEMSIM*(txt$_1$, txt$_2$)* = true, if WMD(txt$_1$, txt$_2$) is greater than a given threshold τ (0.65 in our experiments), false otherwise. We implement ISSEMSIM as a Boolean function with a threshold, because the WMD distances are not accurate enough to assume that the highest similarity is the best one [29], [30]. For example,

two sentences with WMD 0.99 might not be more semantically similar than two sentences with WMD 0.88 [30].

We now define the **semantic matching of events, denoted by** $\sim$. Given a donor test t_D, a recipient test t_R, and two events $e_i \in t_D$ and $e_j \in t_R$, with descriptor d_i and d_j, respectively, we say that $\mathbf{e_i} \sim \mathbf{e_j}$ if one of the following cases holds.

Matching click events: $e_i = \textit{click}(w_i) \wedge e_j = \textit{click}(w_j) \wedge \textsc{IsSemSim}(d_i, d_j) = \textsf{true}$. This is the case of clicks events that execute a similar functionality.

Matching fill events: $e_i = \textit{fill}(w_i, \textit{txt}_i) \wedge e_j = \textit{fill}(w_j, \textit{txt}_j) \wedge \textsc{IsSemSim}(d_i, d_j) = \textsf{true} \wedge \textit{txt}_i = \textit{txt}_j$. This is the case of fill events that execute a similar functionality with the same input.

Matching fill-to-click events: $e_i = \textit{fill}(w_i, \textit{txt}_i) \wedge e_j = \textit{click}(w_j) \wedge \textsc{IsSemSim}(\textit{txt}_i, d_j) = \textsf{true}$. This is the case of a fill event in t_D that can be mapped to an equivalent click event in t_R. For example, entering the value "1" in a calculator app can be mapped to clicking the button with text "1" in another calculator app. The reader should notice that we do not allow the opposite, that is, mapping click events of t_D to fill events of t_R. Otherwise, ADAPTDROID could easily (and incorrectly) map a button click in t_D with an event of A_R that enters the label of the clicked button in an input field.

In the example of Figure 1, ADAPTDROID matches the events in t_D with those in t_R as follows:

$$e_{D1} \sim e_{R1} \quad e_{D3} \sim e_{R1} \quad e_{D3} \sim e_{R4} \quad e_{D4} \sim e_{R5}$$

B. Semantic Information Extraction

This phase executes t_D in A_D to collect the following information, which are required by the next phases.

- **Oracle assertions.** For each oracle assertion in O_t, ADAPTDROID logs both the state of the widgets, when each assertion is checked, and the expected value of the assertion.

- **Events ordering.** Obtaining a meaningful test adaptation that preserves the semantics of t_D may require that some events are executed in a specific order. Conversely, certain events may follow alternative orders without affecting the semantics of the test (such as, the fill events that fill a form). ADAPTDROID identifies such events to avoid unnecessary constraints on the events ordering while generating the adapted test.

To identify the opportunity of re-ordering events, ADAPTDROID checks if each pair of consecutive events e_i and e_{i+1} in t_D could be potentially executed in the opposite order. Let us consider $\ldots S_{i-1} \xrightarrow{e_i} S_i \xrightarrow{e_{i+1}} S_{i+1} \ldots$, which indicates the sequence of states traversed with the execution of events e_i and e_{i+1}. We say that events e_i and e_{i+1} *can be reordered*, denoted by $e_i \rightleftharpoons e_{i+1}$, iff e_{i+1} is enabled in state S_{i-1} and e_i is enabled in state S_{i+1}. We say that an event e that interacts with a widget w is enabled in a state S iff S contains a widget w' with the same xpath of the widget w and w' is interactable.

We define the *cluster of the events that can be arbitrarily reordered* as the set of consecutive events that can be reordered. Formally, given $e_i \rightleftharpoons e_{i+1}, \forall i = j \ldots m \ (1 \leq j < m < n)$, the corresponding cluster is $C = \{e_j, \ldots e_{m+1}\}$. We also say that $\textit{cluster}(e_i) = C, \forall i = j \ldots m+1$. We build the clusters by checking each pair of consecutive events to guarantee a linear time complexity with respect to test length. For instance, t_D in Figure 1 has four clusters with a single event each, indicating that the prescribed order is the only possible one.

To facilitate the definition of the next phases, we introduce the Boolean function $\textsc{isBefore}(e_i, e_j)$ that returns true iff $\textit{cluster}(e_i) \neq \textit{cluster}(e_j) \wedge i < j$ (e_i must strictly precede e_j in t_D), false otherwise.

C. Initial Population Generation

Any evolutionary algorithm starts by generating $\mathcal{P}_0$ the initial population of N individuals [31]. An individual for ADAPTDROID is a test t_R for the recipient application A_R. ADAPTDROID populates $\mathcal{P}_0$ with *NR* randomly-generated tests (to guarantee genetic diversity in $\mathcal{P}_0$) and *NG* tests ($N = NG + NR$) generated with a greedy algorithm that are similar to t_D (to have "good" genetic material for evolution).

ADAPTDROID generates random tests following standard random approaches [2]. Specifically, ADAPTDROID generates a random test t_R by opening/restarting A_R to obtain the initial state S_0, and repeating the following three steps until t_R reaches the maximum length L: (i) it randomly selects an event e_i from those enabled in the current GUI state S_{i-1}; (ii) it appends e_i to t_R; (iii) it executes e_i obtaining the state S_i.

The greedy-algorithm chooses an event e_i among the events that semantically match an event in t_D, and then executes step (ii) and (iii) of the random-algorithm. In details, step (i) of the greedy-algorithm selects an event e_i from the set $\{e_j \ : \ e_j \textit{ is enabled in } S_{i-1} \wedge \exists e_k \in t_D : e_j \sim e_k\}$. If this set is empty, it selects an event at random.

D. Fitness Calculation

At each generation *gen* of the evolutionary algorithm, the *Fitness Calculation* computes a fitness score in [0,1] for each test t_R in $\mathcal{P}_{gen}$. The score characterizes the similarity between t_R and t_D, and guides the exploration of possible test adaptations. ADAPTDROID computes the fitness score by executing each t_R in $\mathcal{P}_{gen}$ and extracting the event descriptors and state transitions. While executing the tests, ADAPTDROID also updates a *GUI model* [3] that encodes the sequence of events that trigger window transitions. The definition of such a model follows the one proposed by Memon et al. [3]. ADAPTDROID uses this model in the *Population Evolution* phase to repair infeasible tests.

We define the fitness function of a test t_R, by considering (i) the number of events in t_D that semantically match the events in t_R (*similar events*), and (ii) the number of assertions in t_D that are applicable to the states reached by t_R (*applicable assertions*). Intuitively, the higher these numbers are the more successful the adaptation is.

Similar Events To compute the number of similar events for each test $t_R \in \mathcal{P}_{gen}$, ADAPTDROID maps the events in t_R to those in t_D using the semantic matching (see Section IV-A). Let $\mathcal{M} \subseteq t_R \times t_D$ denote a binary relation over t_R and t_D, that we call **mapping**, such that each pair of events semantically match. That is, $\mathcal{M}$ is a set of pairs of events $(e_R \in t_R, e_D \in t_D) \ : \ \forall (e_R, e_D) \in \mathcal{M}, e_R \sim e_D$. An event in t_R can be

mapped to multiple events in t_D. For instance, in Figure 1 e_{R1} maps both e_{D1} and e_{D3}. We use $\mathbb{M}$ to denote all the possible mappings between events in t_R and t_D.

Many mappings in $\mathbb{M}$ could be invalid. A mapping $\mathcal{M} \in \mathbb{M}$ is **valid** iff all the following three criteria are satisfied:

1) Injective matching $\mathcal{M}$ does not contain any event in t_R that relates with more than an event in t_D: $\forall e_{RA}, e_{RB} \in t_R$ and $\forall e_D \in t_D$, if $(e_{RA}, e_D) \in \mathcal{M} \wedge (e_{RB}, e_D) \in \mathcal{M}$, then $RA = RB$ (e_{RA} and e_{RB} are the same event). In the example of Figure 1, the mapping $\mathcal{M} = \{(e_{R1}, e_{D3}), (e_{R4}, e_{D3})\}$ is invalid because it does not satisfy this criterion.

2) Valid ordering All events in t_R satisfy the ordering of t_D as extracted in the *Semantic Information Extraction* phase: $\forall (e_{RA}, e_{DA}), (e_{RB}, e_{DB}) \in \mathcal{M}$ if ISBEFORE(e_{DA}, e_{DB})=true, RA<RB (e_{RA} precedes e_{RB} in t_R).

3) Consistent matching Two events in t_D that are associated with the same event descriptor must be matched to consistent recipient events in t_R: $\forall (e_{RA}, e_{DA}), (e_{RB}, e_{DB}) \in \mathcal{M}$ if e_{DA} and e_{DB} have identical descriptors ($d_{e_{DA}} = d_{e_{DB}}$), then also e_{RA} and e_{RB} must have identical descriptors ($d_{e_{RA}} = d_{e_{RB}}$). This constraint avoids mapping two equivalent events in t_D (such as clicking the same button) to different widgets in A_R.

ADAPTDROID selects the valid mapping $\mathcal{M}^\star \in \mathbb{M}$ that maximizes the number of matched events, and uses $\mathcal{M}^\star$ to compute the event similarity between the two tests. Intuitively, $\mathcal{M}^\star$ is the mapping that best captures the similarity of t_R and t_D. More formally, $\mathcal{M}^\star \in \mathbb{M}$ such that $\mathcal{M}^\star$ is valid and $\nexists$ a valid $\mathcal{M} \in \mathbb{M}$:$|\mathcal{M}|$>$|\mathcal{M}^\star|$. $|\mathcal{M}|$ indicates the number of pairs in a mapping $\mathcal{M}$. If there are multiple valid mappings with the highest cardinality, ADAPTDROID selects one randomly. In the example of Figure 1, $\mathcal{M}^\star = \{(e_{R1}, e_{D2}), (e_{R4}, e_{D4}), (e_{R5}, e_{D6})\}$.

Because of the huge number of possible mappings ($2^{|t_R| \cdot |t_D|}$), ADAPTDROID does not enumerate $\mathbb{M}$ and then remove all invalid mappings. Instead, ADAPTDROID efficiently identifies $\mathcal{M}^\star$ by applying the three validity criteria while building $\mathbb{M}$.

Applicable Assertions ADAPTDROID fitness function also considers the number of assertions in t_D that "*can be applied to*" t_R. This is because a good adaptation of the donor test t_D must reach a state of the recipient app with widgets that are compatible with the ones checked by the donor assertions.

Intuitively, an assertion $o \in O_D$ is *applicable* in t_R if o can be applied to at least a state reached after the execution of the last event in the mapping $\mathcal{M}^\star$ (we recall that we only consider assertions at the end of the tests). The applicability of an assertion in a state depends on the existence (or absence) of widgets in the recipient app that are semantically similar to the widgets checked by the donor assertion in the donor app.

ADAPTDROID supports four types of assertions: o_1 = *exists(txt)* and o_2 = *hasText(w,txt)*), and their negative counterparts: $\overline{o_1}$ = *not(exists(txt))* and $\overline{o_2}$ = *not(hasText(w,txt))*.

For the positive assertion types o_1 and o_2, the Boolean function ISAPPLICABLE$(o, \mathcal{M}^\star)$ returns true iff o is *applicable* in the state reached after executing the last event of t_R in $\mathcal{M}^\star$, false otherwise. An assertion o is *applicable* in a state S_i if there exists a widget $w' \in S_i$ such that ISSEMSIM$(d_{w'}, d_o)$ = true, where $d_{w'}$ is the descriptor of the widget w' extracted with the rules in Section IV-A. The descriptor of an assertion of type o_1 = *exists(txt)* is d_{o1} = *txt* , while for type o_2 = *hasText(w,txt)* is $d_{o2} = d_w$.

For the negated assertion types $\overline{o_1}$ and $\overline{o_2}$, we define the ISAPPLICABLE function differently. This is because it is trivial to find a state that does not contain a certain widget/text. Indeed, most of the states traversed by an adapted test satisfy this condition. To better capture the semantics of negated assertions, we force t_R to explicitly move the recipient app from a state that does not satisfy the assertion to a state that satisfies it. Since we check for the absence of a certain widget/text, we also require t_R to satisfy this constraint on the same window. Otherwise, the constraint could be easily satisfied by changing the current window of the app. More formally, given a negated assertion $\overline{o}$, ISAPPLICABLE$(\overline{o}, M^\star)$ returns true iff (i) the positive version of $\overline{o}$ (obtained by removing *not*) is applicable in a state S_i traversed by t_R, (ii) the positive version of $\overline{o}$ is not applicable in a state S_j traversed after the last event in the mapping $M^\star$, (iii) S_i is traversed before S_j, and (iv) both S_i and S_j refer to the same window; false otherwise.

In the example of Figure 1, assertion o_{D1} in t_D verifies that no widget with text "Test" exists. The assertion is applicable to t_R because the last state of t_R does not contain such a widget (o_{D1} is true), the state after the event e_{R4} does (o_{D1} is false), and these two states belong to the same window.

Let $O_D^\star \subseteq O_D$ denote the set of assertions of t_D such that ISAPPLICABLE$(o, M^\star)$ returns true. As such, the cardinality of $O_D^\star$ ($\mid O_D^\star \mid$) measures *the number of assertions successfully adapted to the recipient app*.

$$\textbf{FITNESS-SCORE}(\mathbf{t_R}) = \frac{\mid M^\star \mid + \mid O_D^\star \mid}{\mid t_D \mid + \mid O_D \mid} \in [0; 1]$$

The fitness score is proportional to both the number of events and the number of assertions in t_D. That is, obtaining an applicable assertion contributes as much as successfully adapting an event. The score is a value in [0, 1], with 1 representing a perfect adaptation.

E. Population Evolution

The *Population Evolution* phase combines and mutates the individuals (GUI tests) in the current population $\mathcal{P}_{gen}$ to generate a new population $\mathcal{P}_{gen+1}$ of size N. We follow the classic evolutionary algorithm [32], which works in four consecutive steps: elitism, selection, crossover and mutation.

Elitism ADAPTDROID adds in $\mathcal{P}_{gen+1}$ the elite set E of observed individuals with the highest fitness score ($|E| < N$). This *elitism* process is a standard genetic algorithm step that avoids missing the best individuals during the evolution [32].

Selection ADAPTDROID selects $N/2$ pairs of individuals from $\mathcal{P}_{gen}$ as candidates for the crossover. We use the standard *roulette wheel* [32] selection that assigns at each individual a probability of being selected proportional to its fitness.

Crossover ADAPTDROID scans each selected pair $\langle t_{R1}, t_{R2} \rangle$ and with probability *CP* performs the crossover and with

probability $1-$ *CP* adds the two tests as they are in $\mathcal{P}_{gen+1}$. The crossover of two parents produces two offspring by swapping their events. ADAPTDROID implements a single-point cut crossover [32] as follows. Given a selected pair $\langle t_{R1}, t_{R2}\rangle$, ADAPTDROID chooses two random cut points that split both t_{R1} and t_{R2} in two segments. It then creates two new tests. One concatenating the first segment of t_{R1} and the second segment of t_{R2}. The other concatenating the second segment of t_{R1} and the first segment of t_{R2}.

The crossover likely yields infeasible tests, where executing the first segment leads to a window (W_1) different from the window (W_2) that the first event in the second segment expects. ADAPTDROID repairs these tests by interleaving the two segments with a sequence of events that move from W_1 to W_2. ADAPTDROID identifies such sequence by querying the GUI Model of A_R (see Section IV-D).

Mutation When the crossover terminates ($|P_{gen+1}| = N$), ADAPTDROID mutates the tests in P_{gen+1} with a certain probability, aiming to both add genetic diversity and quickly converge to a (sub)optimal solution. As such, ADAPTDROID uses two mutations types: random and fitness-driven.

Random Mutations mutate the tests in P_{gen+1} with a probability *RM* by applying any of these mutations: (i) adding an event in a random position; (ii) removing a randomly selected event; (iii) adding multiple random fill events in a window containing multiple text fields. The rationale of the last mutation is that forms with several fields might require many generations to be entirely filled out. This mutation speeds up the evolution by filling all the text fields in a single mutation.

Fitness-Driven Mutations mutate a test to improve its fitness score. Each test in P_{gen+1} has a probability *FM* of being mutated using one of these two mutations: (i) removing an event in t_R that does not match (according to $\mathcal{M}^\star$) any event in t_D; (ii) adding an event e_j in t_R such that $e_i \sim e_j$, where e_i is a randomly selected event in t_D that does not match t_R events.

Like crossovers, also mutations could create infeasible tests. ADAPTDROID identifies them by checking if all the events in the mutated tests can be executed in the order prescribed by the test, and fixes the infeasible tests it by removing all non-executable events. Indeed, the fixed test could still have useful genetic material for the evolution [33].

The search for an adapted test keeps evolving and evaluating populations of tests until either a predefined budget expires (# of generations or time) or ADAPTDROID finds a test with fitness one. When the search terminates, ADAPTDROID post-processes the test with the highest fitness score by reducing the test length, and injecting the donor assertions (if possible).

ADAPTDROID reduces the test length by removing one by one the events that are not part of the mapping $\mathcal{M}^\star$ used to calculate the fitness score. After removing an event, ADAPTDROID executes the test and recalculates its fitness. If the fitness decreases, ADAPTDROID restores the event because, even though it did not directly contribute to the fitness value, it enabled other relevant events to be executed. In the $\mathcal{M}^\star$ of the example of Figure 1, events e_{R2}, e_{R3}, e_{R6}, and e_{R7} of t_R do not match any event in t_D, but the post-process keeps them because removing any event reduces the fitness.

If the fitness function finds some assertions in t_D that are applicable to t_R, ADAPTDROID adds them at the end of t_R. In the example of Figure 1, ADAPTDROID injects the assertion o_{D1} = *not(exists("Test"))* at the end of t_R.

V. EVALUATION

We evaluated ADAPTDROID by implementing a prototype tool for Android apps [22]. Our prototype uses the APPIUM 6.1.0 framework [34] to read the GUI states and Android emulators to execute the tests. We evaluated ADAPTDROID considering two research questions:

RQ1: Effectiveness *Can* ADAPTDROID *effectively adapt GUI tests and oracles across similar applications?*

RQ2: Baseline Comparison *Is* ADAPTDROID *more effective than baseline approaches?*

To measure the quality of test adaptations we need human judgment, possibly involving the designers of the donor tests. For this reason, we evaluated ADAPTDROID with a human study that involved four PhD students majoring in Software Engineering, who were not related to this project. We asked each participant to design some donor tests and to evaluate the adaptations produced by ADAPTDROID.

Selecting Subjects and Collecting Donor Tests We selected a total of 32 Android apps (8 donors and 24 recipients) from the Google Play Store by referring to four app categories that represent apps with recurrent functionalities [15]: *Expense Tracking*, *To-Do List*, *Note Keeping*, and *Online Shopping*. We avoided selection biases as follows.

We queried the Google Play Store by searching for each category name. From the list of returned apps, we selected the first two that are free/freemium and do not require login credentials at start-up. Thus, obtaining a total of eight donor apps. For each donor app A_D, we identified three recipient apps by retrieving the list of similar apps suggested in the Google Play Web page of A_D. From this list, we selected the first three apps that were not selected as donors and have the same characteristics described above. This process resulted in 24 pairs $\langle A_D, A_R\rangle$ of donor and recipient apps.

We randomly partitioned the eight donor apps among the testers, by assigning two donor apps of different categories to each tester. In this way, we prevented that a tester could design similar donor tests. We asked each tester to design a Selenium GUI test [35] (with an oracle assertion) to test the main functionality of the app. We left up to the tester to identify the main functionality of the app.

After each tester implemented a donor test, we asked to evaluate whether the test could be adapted to the recipient apps. Each tester evaluated each adaptation on a scale "Fully" (the main functionality of A_R can be tested as in t_D), "Partially" (A_R allows to replicate only some of the operations performed in t_D), "No" (A_R implements no functionality that can be tested as in t_D). Column "$\langle A_D, A_R\rangle$ *Adaptable?*" of Table I reports the responses. The testers deemed fully adaptable 18 pairs of

TABLE I
EVALUATION SUBJECTS AND RESULTS

Tester	Subject description: Donor App (A_D)	$\lvert t_D \rvert$	Recipient App (A_R)	ID	$\langle A_D, A_R \rangle$ adaptable?	ADAPTDROID: $\lvert t_R \rvert$	fitness	gen.	RQ1: Effectiveness: Q_T	# spurious events	# missing events	Q_S	Oracle adapted?	RQ2: Baseline: Random fitness	Random gen.	Basic fitness	Basic gen.
T1	Expense Manager (Expense Tracking)	15	KPmoney	1	Partially	13	**0.64**	33	3	6	0	1.00	No	0.30	79	0.57	50
			Monefy	2	Yes	10	**0.47**	59	4	1	0	1.00	Yes	0.23	3	0.43	98
			Money	3	Yes	15	**0.47**	95	4	0	0	1.00	Partially	0.30	2	0.40	82
	Mirte Notebook (Note Keeping)	10	Xnotepad	4	Partially	7	**0.78**	91	1	4	3	0.50	No	0.36	8	0.77	80
			Color Notes	5	Yes	15	0.24	4	2	9	2	0.75	Partially	0.21	40	**0.27**	9
			Keep Mynotes	6	Yes	2	**0.39**	1	1	1	5	0.14	No	0.33	79	0.33	11
T2	Markushi Manager (Expense Tracking)	16	Spending Tracker	7	Yes	23	**0.74**	59	2	1	8	0.73	Partially	0.72	7	**0.74**	36
			Smart Expenditure	8	Yes	18	**0.41**	25	0	-	-	-	-	0.32	3	**0.41**	16
			Gastos Diarios	9	Partially	6	**0.50**	17	0	-	-	-	-	0.37	19	**0.50**	22
	Bitslate Notebook (Note Keeping)	13	Notes	10	Yes	8	**0.45**	15	3	0	2	0.80	No	0.33	61	0.31	10
			Fast Notepad	11	Yes	10	0.41	86	4	0	2	0.83	Yes	0.23	6	**0.44**	41
			Notepad	12	Yes	12	**0.11**	1	1	10	11	0.15	Yes	**0.11**	1	**0.11**	1
T3	Pocket Universe (To-dolist)	11	Seven Habits	13	No	-	-	-	-	-	-	-	-	-	-	-	-
			Ob Planner	14	No	-	-	-	-	-	-	-	-	-	-	-	-
			Simplest Checklist	15	No	-	-	-	-	-	-	-	-	-	-	-	-
	Aliexpress (Online Shopping)	16	Banggood	16	Yes	11	0.50	43	2	4	1	0.88	No	0.30	11	**0.64**	36
			Light in the box	17	Yes	7	**0.50**	75	1	4	5	0.38	No	0.23	23	0.44	72
			Shein	18	Yes	9	**0.30**	92	0	-	-	-	No	0.17	62	**0.30**	64
T4	Zalando (Online Shopping)	6	Zara	19	Yes	8	**0.42**	19	3	0	0	1.00	No	0.38	32	**0.42**	22
			Romwe	20	Yes	5	**0.54**	5	3	0	1	0.83	No	0.50	1	**0.54**	11
			Yoox	21	Yes	-	-		-	-	-	-	-	-	-	-	-
	Splendo (To-dolist)	10	To Do List	22	Yes	9	**0.63**	59	3	4	6	0.45	Yes	0.38	46	0.55	26
			Tasks	23	Yes	8	**0.46**	32	1	6	5	0.29	No	0.34	59	0.38	10
			Tick Tick	24	Yes	15	0.46	21	1	11	8	0.33	Yes	0.29	7	**0.53**	93

TABLE II
CONFIGURATION PARAMETERS OF ADAPTDROID

name	description	value	name	description	value
τ	threshold for WMD	0.65	N	population size	100
E	# tests for elitism	10	L	max length of the initial tests	$\lvert t_D \rvert$
NR	# initial random tests	90	*NG*	# initial greedy tests	10
CP	crossover prob.	0.40	*RM*	random mutation prob.	0.35
FM	fitness-driven mutation prob.	0.35			

tests (75%) and partially 3 pairs of tests (12.5%). This result confirms the intuition that GUI tests can be adapted across similar applications. Tester T_3 deemed the pairs with ID 13, 14 and 15 as not adaptable, because the test executes functionalities available only in the donor *Pocket Universe* and not in the recipient apps. We asked the four testers to manually adapt the fully and partially adaptable donor tests to the recipient apps.

Running ADAPTDROID We ran ADAPTDROID with the 21 fully and partially adaptable donor tests giving as input the pairs $\langle A_D, A_R \rangle$ and the corresponding manually-written test t_D. We used a popular WMD model trained on a *Google News* dataset (about 100 billion words) [36].

We ran ADAPTDROID with a budget of 100 generations with the configuration parameters values shown in Table II. We selected these values by performing some trial runs and by following basic guidelines of genetic programming [31]. Special considerations can be made for the values τ and L. We chose $\tau = 0.65$ as the threshold for the semantic similarity by evaluating the WMD model on a list of ∼2.5 M synonyms [37]. More specifically, $\tau = 0.65$ is the threshold that achieves the best trade-off between matched synonyms and unmatched pair of randomly selected words. We choose $L = |t_D|$ to obtain initial tests for A_R with a max length proportional to the length of the donor test.

When dealing with the test pair with ID 21, we experienced some compatibility issues between the APPIUM framework and the recipient app A_R, issues that prevented ADAPTDROID generating tests. Thus, we exclude such a pair from our analysis.

The "ADAPTDROID" columns of Table I show information about the returned adapted test t_R (the one with the highest fitness score). Column "$|t_R|$" provides the number of events of the adapted test. Column "*fitness*" shows the fitness score of t_R. ADAPTDROID never reached fitness score 1.0, thus it terminated after 100 generations. Column "*gen.*" shows the generation in which ADAPTDROID produced t_R.

ADAPTDROID completed 100 generations in 24 hours on average, and spent most of this time in executing the generated tests on the emulator. Executing tests is expensive because ADAPTDROID re-installs A_R in the emulator before each test execution to guarantee that each test executes from a clean state. This time could be reduced by running many emulators in parallel or using cloud platforms for mobile testing.

RQ1: Effectiveness We asked the testers to judge the quality of each test case t_R produced with ADAPTDROID for their assigned pairs. We used a score from 0 to 4, where 0 means that t_R is completely unrelated to the donor test semantics, and 4 means that t_R is an adaptation as good as the one that they manually produced (Column "Q_T" of Table I).

In 8 cases out of 20 (40%) the testers evaluated ADAPTDROID adaptations as high quality ($Q_T \geq 3$), with three of which considered perfect adaptations. In three cases (15%), the adapted tests were evaluated as medium quality ($Q_T = 2$). This suggests that in these eleven cases ($Q_T \geq 2$) the fitness function well describes the similarity with the donor test.

We asked the testers to indicate the spurious and missing events in the tests. Columns "*# spurious events*" and "*# missing events*" of Table I report the number of events identified as spurious and missing to obtain a perfect adaptation,

respectively. Column "Q_S" of Table I reports a *structural* quality indicator of the completeness of the matched events: $Q_S = 1 - (\#^{missing}/_{|\overline{t_R}|})$, where $\overline{t_R}$ is the manually adapted test. Q_S ranges in [0, 1], where 0 indicates *no* matching between the events in t_R and $\overline{t_R}$, and 1 indicates a perfect matching. The average of Q_S is f 0.53, indicating that overall ADAPTDROID adapted 53% *true* event matches identified by the testers. There is a moderate correlation between the two quality indicators Q_T and Q_S (Pearson coefficient is 0.89). This confirms that it is important for the testers to adapt a large portion of a test.

Column "*Oracle Adapted?*" in Table I reports whether t_R has an assertion that the tester evaluated to be correct ("Yes"), partially correct ("Partially"), or not applicable in the states reached by t_R ("No"). Testers T_1 and T_2 attributed partial correctness to three adapted oracles because of marginal differences in the expected output. For instance, in the pair with ID 3, the oracle in the donor test checks if a widget with descriptor "expenses" has text "100". The corresponding widget in the recipient test has text "-100", which is semantically equivalent (100 expenses = -100 balance) but syntactically different. Therefore, the oracle was deemed partially correct.

We identify two main issues that limited the effectiveness of ADAPTDROID.

1) Significant differences between A_D and A_R. For instance, in the pair with ID 18, t_D searches in the *Aliexpress* app for a USB drive and adds it to the shopping basket. Tester T_3 manually adapted t_D searching in the *Shein* app for a t-shirt. ADAPTDROID failed to adapt this test to *Shein*, as searching for a USB drive in *Shein* results in an empty search. As another example, in the pairs with ID 23 and 24, t_D adds a task to a pre-existing *work* task list. ADAPTDROID failed to adapt these tests, as the recipient apps do not have a task list.

2) Missed Event Matches. The semantic matching of the descriptors was not always precise due to (i) the unsoundness of WMD; and (ii) the limited semantics information of the event descriptors. For example, some of the considered apps have image buttons with file name "fabButton.png", which does not describe the semantics of the widget.

The results of our study are promising: ADAPTDROID produced eleven good quality test adaptations between apps with very different GUIs. In the experiments, we configure ADAPTDROID to report all adaptations. We can improve the quality of the generated output, by reporting only adapted tests that reach a minimum fitness score.

RQ2: Baseline Comparison We compare ADAPTDROID with two baseline approaches (i) RANDOM, to empirically assess the effectiveness of the evolutionary algorithm of ADAPTDROID; (ii) ADAPTDROID-BASIC, a restrained version of ADAPTDROID without the fitness-driven mutations and greedy-matching initialization, to assess their impact to the overall effectiveness.

We obtained RANDOM from ADAPTDROID by (i) replacing the roulette-wheel selection with random selection, (ii) randomly generating the tests in the initial population (*NR* $= 100$ and *NG* $= 0$, Table II), (iii) setting the probability of the fitness-driven mutations to zero (*FM*$= 0.0$ Table II), and (iv) disabling elitism. As such, RANDOM carries population initialization and evolution completely random. We opted to use a random variant of ADAPTDROID rather than an existing random generator [2], for a meaningful evaluation. With an existing random generator, we cannot ensure that the differences are due to the search strategy and not to differences in other implementation details, such as the events and inputs types considered by the tools.

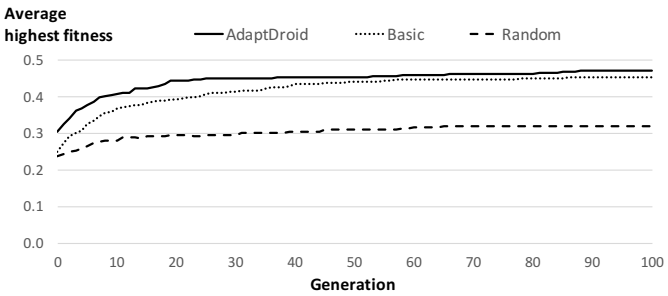

Fig. 3. Average fitness growth.

We obtained ADAPTDROID-BASIC from ADAPTDROID by applying only the modifications (ii) and (iii) described above.

We ran RANDOM and ADAPTDROID-BASIC with a budget of 100 generations as ADAPTDROID. The last four columns in Table I show the fitness score of the fittest test and the generation that created it (with the highest fitness value among the tools in bold). ADAPTDROID consistently achieves a higher fitness than RANDOM, and the same fitness only in one case (ID 22). The difference between the tools is statistically significant: a two-tailed t-test returns a p-value of 0.0002. ADAPTDROID achieves an average fitness of 0.48, while ADAPTDROID-BASIC of 0.45. This shows a difference albeit small of the fitness score.

Figure 3 illustrates the gain of ADAPTDROID over the baseline approaches, by plotting the highest fitness score per generation averaged over the 20 adaptations. The plot highlights three important aspects:

I.- the fittest test of the greedily generated initial population ($\mathcal{P}_0$) of ADAPTDROID has a much lower fitness score than the final score. This shows that the greedy algorithm used to initialize the population is inadequate, thus motivating the use of an evolutionary approach.

II.- the highest fitness per generation of ADAPTDROID steadily increases while the fitness of RANDOM saturates much faster. This demonstrates that ADAPTDROID evolutionary approach is effective in exploring the search space, and it confirms our hypothesis that ADAPTDROID generates GUI tests that can hardly be generated at random.

III.- ADAPTDROID and ADAPTDROID-BASIC reach similar fitness, but ADAPTDROID reaches it faster. This indicates that the greedy-match initialization and the fitness-driven mutations help to converge faster to the fittest test.

Threats to Validity A main threat to the external validity concerns the generalizabilty of a small set of adaptations. The scale of the experiment is limited, due to the cost of involving human testers. However, it is comparable to the ones of the main related approaches [19], [20].

Another threat relates to the selection of testers. The four testers have testing experience, but they are not the developers of the apps. We mitigate this issue by letting the testers get familiarity with the apps before asking them to design the tests.

A final threat relates to the statistical significance of the results. Since the evolutionary algorithm is inherently stochastic, multiple runs may yield different results. Since the evaluation of the results involved human participants, who can only evaluate few tests, we ask them evaluate a single result of ADAPTDROID.

VI. RELATED WORK

GUI Test Generation Existing generators of GUI tests [38], [39] have two major limitations: (i) lack of domain knowledge [40], and thus they may generate either unrealistic or semantically meaningless GUI tests [8], (ii) lack of automated oracles, and thus they are able to only detect crashes or exceptions [10], [11]. ADAPTDROID addresses these limitations by generating semantic GUI tests and oracles adapted from manually-written GUI tests of similar apps.

Researchers have exploited usage data to improve GUI test generation [40]–[43]. For example, POLARIZ [40] and MONKEYLAB [41] generate Android tests using GUI interaction patterns extracted from app usage data. Differently, ADAPTDROID fully adapts existing tests across apps maintaining the same semantics of the donor test.

Similarly to ADAPTDROID, the GUI test generators AUGUSTO [14] and APPFLOW [15] exploit commonalities among GUI applications. However, they do not aim to adapt tests across applications nor leverage existing GUI tests. Instead, they rely on a set of manually-crafted GUI interaction patterns. Moreover, APPFLOW recognizes common widgets using a semi-automated machine learning approach. Conversely, ADAPTDROID matches GUI events without requiring human intervention.

GUI Test Adaptation CRAFTDROID [19] and APPTESTMIGRATOR [20], [44] are the first attempts to adapt GUI tests across mobile apps. Both approaches explore a statically computed GUI model of the *recipient app* to "greedly" find a sequence of events that maximizes the semantic similarity with the events of the donor tests. Similarly to ADAPTDROID, they extract event descriptors from the GUI and match them across applications using word embedding [21]. However, ADAPTDROID differs substantially from these two techniques.

ADAPTDROID shares the overall objective of adapting tests across applications, but introduces substantial novelties. Both CRAFTDROID and APPTESTMIGRATOR use a greedy algorithm, which resembles the one that ADAPTDROID uses to generate the initial population $\mathcal{P}_0$. ADAPTDROID uses an evolutionary algorithm to improve an initial set of greedy-matched tests. As discussed in Section V, ADAPTDROID evolutionary approach largely improves over a greedy algorithm. Indeed, APPTESTMIGRATOR and CRAFTDROID explore *a single* test adaptation, and do not consider alternative sequences of random events that could yield to a better adaptation. The ADAPTDROID evolutionary approach explores *many possible* test adaptations to find the sequence of events that yields the best adaptation. As such, ADAPTDROID can be used to improve the tests adapted with APPTESTMIGRATOR or CRAFTDROID.

GUI Test Repair Test repair techniques fix tests that become invalid during software evolution [45]–[50]. These techniques assume that most widgets remain unmodified between versions of the same app [45], [51], and do not address the core challenge of semantically matching widgets across apps.

VII. CONCLUSIONS

This paper presents ADAPTDROID an evolutionary technique to adapt test cases across mobile apps that share similar functionalities. Our empirical evaluation indicates that ADAPTDROID can adapt useful and non-trivial GUI tests across semantically similar apps with very different GUIs. This confirms that formulating the test adaptation problem with an evolutionary approach is a viable solution. An important future work is to reduce the computational cost of ADAPTDROID by implementing a distributed version of the tool that executes the evolutionary algorithm on the cloud. Indeed, one of the key advantages of evolutionary algorithms is that are easily parallelizable. Another interesting future work is to extend ADAPTDROID to adapt tests across different platforms, for instance, to adapt GUI tests from Mobile to Web applications.

REFERENCES

[1] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing Combinatorics in GUI Testing of Android Applications," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 559–570.

[2] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '13. ACM, 2013, pp. 224–234.

[3] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of The Working Conference on Reverse Engineering*, ser. WCRE '03. IEEE Computer Society, 2003, pp. 260–269.

[4] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated System Input Feneration for Android Applications," in *Proceedings of the International Symposium on Software Reliability Engineering*, ser. ISSRE '15. IEEE Computer Society, 2015, pp. 461–471.

[5] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '17. ACM, 2017, pp. 245–256.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of Android applications," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '12. ACM, 2012, pp. 258–261.

[7] W. Yang, Z. Chen, Z. Gao, Y. Zou, and X. Xu, "GUI testing assisted by human knowledge: Random vs. functional," *J. Syst. Softw.*, vol. 89, pp. 76–86, 2014.

[8] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*, ser. SOS '11. IEEE, 2011, pp. 13–24.

[9] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '16. IEEE Computer Society, 2015, pp. 429–440.

[10] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '16. IEEE Computer Society, 2016, pp. 33–44.

[11] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, "ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '19. IEEE Computer Society, 2019, pp. 128–139.

[12] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE'16, 2016, pp. 987–992.

[13] A. Rau, J. Hotzkow, and A. Zeller, "Efficient gui test generation by learning from tests of other apps," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE Poster '18. ACM, 2018, pp. 370–371.

[14] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 280–290.

[15] G. Hu, L. Zhu, and J. Yang, "AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests," in *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '18. ACM, 2018, pp. 269–282.

[16] M. Linares-Vásquez, A. Holtzhauer, and D. Poshyvanyk, "On automatically detecting similar android apps," in *Proceedings of the International Conference on Program Comprehension*, ser. ICPC '14. IEEE Computer Society, 2016, pp. 1–10.

[17] I. J. M. Ruiz, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A Large-Scale Empirical Study on Software Reuse in Mobile Apps," *IEEE Software*, vol. 31, no. 2, pp. 78–86, 2014.

[18] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "On Identifying and Explaining Similarities in Android Apps," *J. Comput. Sci. Technol.*, vol. 34, no. 2, pp. 437–455, 2019.

[19] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE'34. IEEE Computer Society, 2019, pp. 42–53.

[20] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE'19. IEEE Computer Society, 2019, pp. 54–65.

[21] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the International Conference on Neural Information Processing Systems*, ser. NIPS '13, 2013, pp. 3111–3119.

[22] L. Mariani, M. Pezzè, V. Terragni, and D. Zuddas, "Adaptddroid tool and experimental data," https://drive.google.com/drive/folders/1NVxoYQZRa5ZFwbnq2QJ4_xGJ-VZci_oX?usp=sharing.

[23] A. Dix, "Human-computer interaction," in *Encyclopedia of database systems*. Springer, 2009, pp. 1327–1331.

[24] F. Song, Z. Xu, and F. Xu, "An xpath-based approach to reusing test scripts for android applications," in *Web Information Systems and Applications Conference (WISA), 2017 14th*, ser. WISA '17. IEEE Computer Society, 2017, pp. 143–148.

[25] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[26] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '14. IEEE Computer Society, 2014, pp. 183–192.

[27] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, ser. ACL '14. Association for Computational Linguistics, 2014, pp. 55–60.

[28] G. Becce, L. Mariani, O. Riganelli, and M. Santoro, "Extracting widget descriptions from guis," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, ser. FASE '12. Springer, 2012, pp. 347–361.

[29] J. Turian, L. Ratinov, and Y. Bengio, "Word representations: a simple and general method for semi-supervised learning," in *Proceedings of the 48th annual meeting of the association for computational linguistics*. Association for Computational Linguistics, 2010, pp. 384–394.

[30] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger, "From word embeddings to document distances," in *Proceedings of the International Conference on International Conference on Machine Learning*, ser. ICML '15, 2015, pp. 957–966.

[31] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[32] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[33] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[34] "Appium," https://github.com/appium.

[35] "Selenium," https://www.seleniumhq.org.

[36] "W2vec pre-trained model," https://code.google.com/archive/p/word2vec/.

[37] G. Ward, "Moby thesaurus," http://moby-thesaurus.org, Accessed: 2018-06-03.

[38] S. Zein, N. Salleh, and J. Grundy, "A Systematic Mapping Study of Mobile Application Testing Techniques," *Journal of Systems and Software*, vol. 117, pp. 334–356, 2016.

[39] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated Testing of Android Apps: A Systematic Literature Review," *IEEE Trans. Reliability*, vol. 68, no. 1, pp. 45–66, 2019.

[40] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '17. IEEE Computer Society, 2017, pp. 16–26.

[41] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *Proceedings of the Working Conference on Mining Software Repositories*, ser. MSR '15. IEEE Computer Society, 2015, pp. 111–122.

[42] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, D. Hao, G. Huang, and F. Feng, "Mining usage data from large-scale android users: challenges and opportunities," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16, 2016, pp. 301–302.

[43] A. M. Fard, M. MirzaAghaei, and A. Mesbah, "Leveraging Existing Tests in Automated Test Generation for Web Applications," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '14, 2014, pp. 67–78.

[44] F. Behrang and A. Orso, "Poster: Automated test migration for mobile apps," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE Poster '18. ACM, 2018, pp. 384–385.

[45] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "Sitar: Gui test script repair," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 170–186, 2016.

[46] A. M. Memon, "Automatically repairing event sequence-based gui test suites for regression testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 2, p. 4, 2008.

[47] A. Memon, A. Nagarajan, and Q. Xie, "Automating regression testing for evolving gui software," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 1, pp. 27–64, 2005.

[48] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving gui applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '13. ACM, 2013, pp. 45–55.

[49] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '12. IEEE Computer Society, 2012, pp. 231–240.

[50] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '17. IEEE Computer Society, 2017, pp. 161–171.

[51] F. Behrang and A. Orso, "Test migration for efficient large-scale assessment of mobile app coding assignments," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '18. ACM, 2018, pp. 164–175.