FrUITeR – A Framework for Evaluating UI Test Reuse

Anonymous Author(s)*

ABSTRACT

UI testing is tedious and time-consuming due to the manual effort required. Recent research has explored opportunities for reusing existing UI tests for an app to automatically generate new tests for other apps. However, the evaluation of such techniques currently remains manual, unscalable, and unreproducible, which can waste effort and impede progress in this emerging area. We introduce *FrUITeR*, a framework that automatically evaluates UI test reuse in a reproducible way. We apply *FrUITeR* to existing test-reuse techniques on a uniform benchmark we established, resulting in 11,917 test reuse cases from 20 apps. We report several key findings aimed at improving UI test reuse that are missed by existing work.

1 INTRODUCTION

Writing UI tests is tedious and time-consuming [23, 26], increasingly driving the focus toward automated UI testing [22]. However, existing work tends to target tests that yield high code coverage, rather than *usage-based* tests that explore an app's functionality, e.g., *sign-in*, *purchase*, *search*, etc. Developers heavily rely on usage-based tests [26], but currently have to write them manually [22, 26].

To reduce the manual effort of writing usage-based tests, recent research has explored reusing existing tests in a *source app* to generate new tests automatically for a *target app* [20, 21, 23, 25, 28]. The guiding insight is that different apps expose common functionalities via semantically similar GUI elements. This suggests that it is possible to reuse existing UI tests across apps—in effect generating the tests *automatically*—by *mapping* similar GUI elements.

Four recent techniques have targeted usage-based test reuse across Android apps [20, 21, 23, 25].[1] While these techniques have shown promise, we have identified five important limitations that hinder their comparability, reproducibility, and reusability. In turn, this can lead to duplication and wasted effort in this emerging area.

❶ The metrics applied to date evaluate whether GUI events from a source app are correctly transferred to a target app, but do not consider *whether the transferred tests are actually useful*. It is possible that events are transferred correctly, but the generated test is "wrong". This can be, e.g., because a generated test is missing events and thus not executable. Moreover, the metrics used in existing work are not standardized even when evaluating related aspects of different techniques, making it difficult to compare the techniques.

❷ Each existing technique's *evaluation process requires significant manual effort*: every transferred event in each test must be inspected to determine whether the transfer is performed correctly. This imposes a practical limit on the number of tests that can be evaluated. For instance, the authors of ATM [21] had to restrict their comparison with GTM [20] to a randomly selected 50% of the possible source-target app combinations due to the task's scale.

❸ There are *no standardized guidelines for conducting the manual inspections*, making the evaluation results biased and hard to reproduce. For instance, ATM's authors acknowledge the possibility of mistakes in the manual process [21]. Such mistakes are currently hard to locate, verify, or eliminate by other researchers.

❹ Existing techniques are *designed as one-off solutions and evaluated as a whole*. This makes it difficult to isolate and compare their relevant components. For instance, GTM [20], ATM [21], and CraftDroid [25] all contain functionality to compute a "similarity score" between two GUI elements, but it is unclear which of those specific components performs best against the same baseline. This impedes subsequent research that could benefit from identifying components that should be reused and/or improved.

❺ Existing techniques make *different assumptions that hinder their comparison*. For instance, GTM [20] and ATM [21] require access to apps' code, and cannot be directly compared with techniques evaluated on close-sourced apps. Similarly, AppFlow [23] requires its tests to be written in a special-purpose language it defines, and cannot transfer tests used in other techniques.

To address limitations 1–3, as well as limitation 4 in part, we have developed *FrUITeR*, a ***Fr***amework for evaluating ***UI Te***st ***R***euse. *FrUITeR* consists of three key elements: a set of *new evaluation metrics* that consolidate the metrics used by existing techniques and expand them to measure important aspects that are currently missed; *two baseline UI test-reuse techniques* that establish the lower- and upper-bounds for the evaluation metrics; and *an automated workflow* that *modularizes* UI test-reuse functionality and significantly reduces the manual effort. With *FrUITeR*, one can automatically evaluate components of UI test-reuse techniques on apps and tests of interest against the same baseline, thus opening the possibility of in-depth studies in this area at a large-scale.

To fully address limitation 4, as well as limitation 5, we have extracted the core components from existing techniques and established a benchmark for evaluating and comparing them. Our benchmark currently contains 20 subject apps with 239 test cases, involving 1,082 GUI events. This benchmark is used by *FrUITeR* to evaluate side-by-side the extracted components and the two baseline components we developed, yielding 11,917 test-reuse instances.

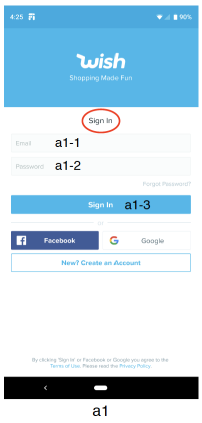
The results obtained by *FrUITeR* revealed several important findings. For example, we have been able to pinpoint specific trade-offs between ML-based (e.g., AppFlow) and similarity-based (e.g., ATM) techniques. We have also identified scenarios that may seem counter-intuitive, such as the fact that manually writing tests requires less effort than attempting automated transfer in certain cases. Finally, performing evaluations on a much larger data corpus allowed us to refute some conclusions reached in prior work.

This paper makes the following contributions. ❶ We develop *FrUITeR* to automatically evaluate UI test reuse with an expanded set of metrics as compared to existing work, and two baseline techniques that help to provide the lower- and upper-bounds of UI test reuse in a given scenario. ❷ We identify and extract the core components from existing test-reuse techniques, enabling their fair

[1] Rau el al. recently proposed a test-reuse technique for web applications [28]. In this paper, we focus on Android apps due to the availability of a larger number of existing techniques to evaluate, although in principle our work is not limited to Android.

comparison. ❸ We establish a reusable benchmark with standardized ground truths that facilitates the reproducibility of UI test-reuse techniques' evaluation and comparison. ❹ We use *FrUITeR* to conduct a side-by-side evaluation of the state-of-the-art test-reuse techniques, uncovering several needed improvements in this area. ❺ We make *FrUITeR*'s implementation and all data artifacts publicly available [12], directly fostering future research.

Section 2 introduces a representative example, relevant terminology, and related work. Section 3 describes *FrUITeR*'s key requirements and Section 4 its design. Section 5 summarizes *FrUITeR*'s implementation and instantiation. Section 6 reports on our evaluation and discusses our key findings. Section 7 concludes the paper.



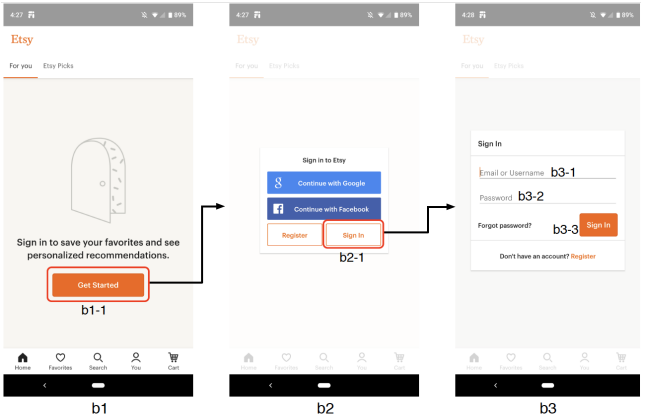


Figure 1: *sign-in* tests for Wish (a1) and Etsy (b1–b3).

2 BACKGROUND AND RELATED WORK

In this section, we introduce a motivating example and relevant terminology, followed by an overview of the strategies pursued by existing work and how they have been evaluated to date.

2.1 Motivating Example and Terminology

Figure 1 shows the screenshots of the *sign-in* process of two popular shopping apps: Wish (left) and Etsy (right). Each screen is labeled with an identifier, e.g., a1 is the first screen of Wish. In each screen, there may be one or more *actionable* GUI elements with which end-users can interact based on the associated actions. For instance, the "Sign In" button in screen a1 (a1-3) is associated with a click action. Actionable elements and their associated actions embody GUI events (defined below). By contrast, the label "Sign In" that is circled in screen a1 is a *non-actionable* GUI element.

As an illustration, assume that Wish's *sign-in* test exists and our goal is to automatically transfer it to Etsy. The relevant actionable GUI elements in this *sign-in* example are labeled and will be used to describe the following key terms used throughout the paper.

GUI Event, or event in short, is a triple comprising (1) an *actionable* GUI element, (2) an associated action, and (3) an optional input value (e.g., user input for a text box). We reuse this definition from existing work [20, 21, 25]. For simplicity, we use the label of a GUI element (e.g., a1-1) to refer to the GUI event triple.

Canonical Event is an abstracted event that captures a category of commonly occurring events. An example canonical event may be AppSignIn, and it would correspond to the a1-3 and b3-3 from Figure 1, as well as similar events from other apps, such as Log In.

Usage-Based Test exercises a given functionality in an app, such as *sign-in*. A usage-based test, or test in short, consists of a sequence of GUI events. For instance, Figure 1 highlights the *sign-in* test in Wish (left) as the event sequence { a1-1, a1-2, a1-3 }.

Source App is the app with known tests that can be transferred to other apps with similar usage. For instance, Wish is a source app with the *sign-in* test that can potentially be transferred to other apps with *sign-in* functionality. **Target App** is the app to which one aims to transfer existing tests. A target app can reuse the tests from multiple source apps; at the same time, it can serve as a source app to other target apps if it contains known tests. Both source apps and target apps are used extensively in existing work [20, 21, 25].

Source Test is an existing test for a given source app that should be transferred to a target app to generate a **Transferred Test**. **Ground-Truth Test** is an existing test for a target app that is used to evaluate whether the transferred test is correct (i.e., whether the two tests match). **Source Event**, **Transferred Event**, and **Ground-Truth Event** refer to the GUI events that belong to the source test, transferred test, and ground-truth test, respectively.

Ancillary Event is a special type of transferred event that is not directly mapped from a source event, but is added in order to reach certain program states in the target app. For example, b1-1 and b2-1 from Figure 1 may need to be added as ancillary events in order to reach Etsy's *sign-in* screen b3; such events do not exist in the source test corresponding to Wish's screen a1.

Null Event is an event that should have been mapped from a source event, but was not identified as such by a given test-reuse technique. Thus, the null event does not exist in the transferred test, but it has a corresponding source event from which it maps. This could be because of (1) a test-reuse technique's inaccuracy or (2) the difference in app behaviors. An example of the latter would be the inability to map Etsy's events b1-1 and b2-1 to Wish in Figure 1.

2.2 Strategies Explored to Date

Four recent techniques [20, 21, 23, 25] have targeted UI test reuse in Android. The shared core concern of these techniques is to correctly map the GUI events from a source app to a target app. In the example from Figure 1, the source test *sign-in* in Wish comprises the event sequence {a1-1, a1-2, a1-3}. By mapping GUI events in this test from Wish to Etsy as {a1-1 → b3-1, a1-2 → b3-2, a1-3 → b3-3}, a new *sign-in* test for the target app, Etsy, is generated as {b3-1, b3-2, b3-3}.

The existing techniques can be classified into two main categories, based on how they map GUI events across different apps: **AppFlow** [23] is an ML-based, while **CraftDroid** [25], **GTM** [20], and **ATM** [21] are similarity-based techniques. We have abstracted the two categories and their underlying workflows through studying the similarities and differences across the existing techniques.

ML-based techniques learn a classifier from a training dataset of different apps' GUI events based on certain features, such as text,

element sizes, and image recognition results of graphical icons. The classifier is used to recognize *app-specific* GUI events and map them to *canonical* GUI events used in a test library, so that app-specific tests can be generated by reusing the tests defined in the test library.

Similarity-based techniques define their own algorithms to compute a *similarity score* between pairs of GUI events in a source app and a target app based on the information extracted from the two apps, such as text, element attributes, and Android Activity/Fragment names. The similarity score is used to determine whether there is a match between each GUI event in the source app and the one in the target app based on a customizable *similarity threshold*. For example, a1-3 in Wish (left) from Figure 1 is likely to have a higher similarity score with b3-3 than with other GUI events in Etsy (right). In that case, a1-3 in Wish will be mapped to b3-3 in Etsy. Another important component in similarity-based techniques is the *exploring strategy*, which determines the order of computing the similarity score between the GUI events in the source and target apps. The target app's events that are explored earlier usually have a higher chance of being mapped.

2.3 Existing Evaluation Metrics

To evaluate their test-reuse strategies, existing techniques have focused on the *accuracy* of the *GUI event mapping*. This section overviews the metrics they applied, which guided us in defining the expanded set of *FrUITeR*'s metrics (see Section 4.1.1). Note that the exact definitions of existing metrics were not provided in the prior publications [20, 21, 23, 25]; we had to separately contact the authors of each technique to obtain the details introduced below.

AppFlow [23] is an ML-based technique that maps app-specific events to canonical events using a classifier as discussed earlier. AppFlow's classifier is evaluated with the standard *accuracy* metric [27], indicating the percentage of the correctly-classified GUI events among all the GUI events being classified. Correctly-classified GUI events include two cases: (1) the app-specific events that are mapped to the correct canonical events (true positive); and (2) the app-specific events that are not mapped to any canonical events and such canonical events do not exist (true negative).

CraftDroid [25] is a similarity-based technique. After the transfer of events from a source app to events in a target app, CraftDroid's authors manually identify three cases: (1) *true positive* (TP) occurs when the transferred event is the same as the one obtained during a manual transfer; (2) *false positive* (FP) occurs when the transferred event is different; and (3) *false negative* (FN) occurs when CraftDroid fails to find a matching event, while the manual transfer succeeds. *Precision* and *recall* are then calculated based on the three cases. It is important to note that CraftDroid's FP includes both the incorrectly transferred events and the newly added *ancillary events* (if any), which is different from the FP case defined in other techniques. We further illustrate this in Section 4.1.

ATM [21] and **GTM** [20] are also similarity-based techniques, and ATM is a direct enhancement of GTM by the same authors. Similarly to CraftDroid, the authors manually inspect the transferred results of each source event and identify four cases: (1) *correctly matched* means the source event is mapped to the correct event in the target app (TP); (2) *incorrectly matched* means the source event is mapped to the wrong event in the target app (FP); (3) *unmatched (!exist)* means the source event is not mapped to any events and no such events exist in the target app (TN); and (4) *unmatched (exist)* means the source event is not mapped to any events although the matching event exists in the target app (FN). Unlike CraftDroid, ATM/GTM do not calculate the precision or recall, but present the raw percentages of each of the four cases.

3 *FrUITeR*'s PRINCIPAL REQUIREMENTS

This section elaborates on the key limitations of current test-reuse techniques and their evaluation processes. These limitations serve as the foundation of five requirements we focused on in *FrUITeR*'s design (Section 4) and instantiation (Section 5).

Prior to developing *FrUITeR*, we investigated the existing techniques and their evaluations [20, 21, 23, 25] in depth. Beyond consulting the available publications, we also studied the techniques' implementations and produced artifacts [1, 3, 5, 8], and engaged their authors in at times extensive discussions to obtain missing details and resolve ambiguities. In the end, we identified five limitations that are likely to hinder future advances in this emerging area. We base *FrUITeR*'s principal requirements on these limitations.

Req$_1$ — *Metrics used by* FrUITeR *to evaluate test-reuse techniques shall be standardized and reflect practical utility.* — Existing techniques are evaluated with different, and differently applied, metrics (recall Section 2.3), which harms their side-by-side comparison. More importantly, all techniques to date have focused on whether GUI events from a source app are correctly transferred to a target app, without considering whether the transferred *tests* are actually meaningful and applicable in the context of the target app. It is thus possible that all GUI events are mapped correctly, but the transferred test cannot be applied, e.g., due to missing ancillary events (recall Section 2.1). None of the existing techniques are able to identify such scenarios; *FrUITeR* must be able to do so.

Req$_2$ — FrUITeR*'s workflow shall reduce the required manual effort and thus scale to larger numbers of apps and tests than possible with current test-reuse techniques.* — Existing techniques' evaluation processes require significant manual effort to inspect every transferred event in each test. For example, ATM [21] was evaluated on 4 app categories, where each category, in turn, consisted of only 4 apps. On average, each app had 10 tests to be transferred and each test had 5 events. Within each app category, ATM transferred the tests of each app to the remaining 3 apps, resulting in 48 source-target app pairs in total. For each app pair, ATM's authors had to manually inspect an average of 50 transferred events (10 tests × 5 events), i.e., 2,400 events in total. This is why they were forced to restrict their comparison with GTM [20] to a randomly selected half of possible source-target app pairs. *FrUITeR* must address this shortcoming by providing a more scalable evaluation workflow that requires markedly less manual effort.

Req$_3$ — *Evaluation results produced by* FrUITeR*'s shall be reproducible.* — As discussed in Section 2.3, the current techniques' evaluation results depend on identifying the case to which each transferred event belongs (e.g., *correctly matched*, *false positive*, etc.). Such "ground-truth mappings" are determined manually. However, there are no standard guidelines for conducting inspections, making the results potentially biased and unreproducible. In Figure 1's example, it is debatable whether {a1-1 → b3-1} is correct because

a1-1 only takes the user's email, while b3-1 takes both the email and username. ATM's authors also acknowledge the possibility of mistakes in the manual process [21]. More importantly, any such mistakes are hard to locate or verify by other researchers because the results of manual inspection and the ground-truth mappings on which they are based are recorded in ad-hoc ways. Thus, to facilitate future research in this area, the evaluation results produced by *FrUITeR* must be reproducible, with a ground-truth representation that can be independently verified, reused, and modified.

Req$_4$ — *Test-reuse capabilities incorporated and evaluated by* FrUITeR *shall be modularized.* — Despite providing similar functionality, existing test-reuse techniques are designed as one-off solutions and evaluated as a whole. This makes it difficult to reuse or compare their relevant components. In turn, it invites duplication of effort and introduces the risk of missed opportunities for advances by other researchers, and even by the techniques' own developers. To address this problem, *FrUITeR* must modularize each test-reuse artifact it evaluates, allow its independent (re)use, and associate the obtained evaluation results with the appropriate artifacts.

Req$_5$ — *Benchmarks provided and applied by* FrUITeR *shall be reusable.* — Existing test-reuse techniques have been evaluated using different benchmark apps and tests, additionally hampering their comparison. In fact, only three subject apps were shared by two of the techniques (AppFlow [23] and CraftDroid [25]) in their evaluations. The underlying reason is the different assumptions made by the techniques. For instance, GTM and ATM rely on the Espresso testing framework [6] that requires the apps' source code. As another example, AppFlow's tests are written in a special-purpose language based on Gherkin [13] and cannot be reused by techniques that capture tests in other languages (e.g., Java, used by ATM and GTM). Thus, *FrUITeR* must establish a set of uniform benchmarks with reusable apps and tests that can serve as the foundation for evaluating and comparing solutions in this area.

4 *FrUITeR*'s DESIGN

This section presents *FrUITeR*'s design, with a focus on two features that address requirements Req$_1$, Req$_2$, Req$_3$, and partially Req$_4$: new evaluation metrics and an automated, modular workflow. We also introduce two novel test-reuse techniques to serve as baselines for bounding the existing techniques' evaluation results.

4.1 *FrUITeR*'s Metrics

To address Req$_1$, *FrUITeR* incorporates a pair of evaluation metrics: (1) *fidelity* focuses on how correctly the *GUI events are mapped* from a source app to a target app; (2) *utility* measures how useful the *transferred tests* are in practice.

4.1.1 ***Fidelity Metrics.*** As explained in Section 2.3, fidelity of the mapping has been the main focus of existing techniques, but the previous metrics have been used inconsistently.[2] To form a fair playground for comparing test-reuse techniques, we investigated existing metrics in-depth. We did so by consulting available documentation and discussing the metrics with the authors of all four techniques. We standardized this information into a *comprehensive* set of fidelity metrics in *FrUITeR*, as shown in Table 1.

Table 1: Fidelity metrics as used in AppFlow [23], CraftDroid [25], ATM [21], GTM [20], and *FrUITeR*.

	True Pos. (TP)	False Pos. (FP)	True Neg. (TN)	False Neg. (FN)	Accuracy	Precision	Recall
AppFlow	*anon*	*anon*	*anon*	*anon*	Accuracy	*dnc*	*dnc*
CraftDroid	TP	FP1 | FP2	*none*	FN	*none*	Precision	Recall
ATM/ GTM	Correctly Matched	Incorrectly Matched	Unmatched (!exist)	Unmatched (exist)	*dnc*	*dnc*	*dnc*
FrUITeR	Correct	Incorrect	NonExist	Missed	Accuracy	Precision	Recall

Table 1 presents the fidelity metrics used across the different test-reuse techniques, and their relationship to the standard metrics as defined in literature [27]. Each row shows a mapping from the names for the metrics used by each technique to the typical fidelity metrics' names indicated in the header. "*anon*" cells represent metrics that are not reported by a technique, but are used internally to calculate other metrics that are reported. "*dnc*" cells represent metrics that are not calculated by a given technique, but can be determined based on other metrics used. Finally, "*none*" cells represent cases where a metric is not used by a technique and cannot be calculated from the available information. *FrUITeR* covers all seven metrics, changing several metrics' names to better reflect their application to of test reuse, as will be further discussed below.

Recall from Section 2.3 that CraftDroid's FP category covers two cases: FP1 corresponds to Incorrectly Matched events in ATM/GTM and Incorrect in *FrUITeR*; FP2 corresponds to the ancillary events that are not considered by other techniques. *FrUITeR* also excludes the ancillary events from its Incorrect category because they can be benign or even needed (e.g., b1-1 and b2-1 from Figure 1), and do not reflect the fidelity of the GUI event mapping. For instance, if ancillary events were considered to be False Positives, a large number of them would result in a low Precision for the GUI event mapping. However, this would not be a meaningful measure since the ancillary events are not mapped from the source app. Such events are thus not relevant to the mapping's *fidelity*, but should be considered by the *utility* metrics, introduced next.

4.1.2 ***Utility Metrics.*** *FrUITeR* incorporates two novel *utility* metrics to indicate how useful a *transferred test* is in practice. These metrics are needed because a high-fidelity GUI event mapping does not guarantee a successfully transferred test, or vice versa. For instance, a target app's ground-truth test may contain *ancillary events* not covered by source events, making it impossible to generate a "perfect" test by event mapping alone. On the flip side, a low-fidelity mapping may accidentally generate a test identical to the ground-truth test. Thus, it is important to measure the utility with respect to the ground-truth test independently of event mapping's fidelity.

To this end, we first define an *effort* metric, to measure how close the transferred test is to the ground-truth test, by calculating the two tests' Levenshtein distance [24]. Levenshtein distance is widely used in NLP to measure the steps needed to transform one string into another. In our case, each step is defined as the *insertion*, *deletion*, or *substitution* of an event in the transferred test.

Secondly, we define a *reduction* metric, to assess the manual effort saved by the generation of the transferred test compared to writing the ground-truth test from scratch:

Reduction = (#gtEvents − Effort) ÷ #gtEvents.

Note that the value of *reduction* may be negative, if transforming

[2] Existing publications in this area have referred to some of these as "accuracy" metrics. We use "fidelity" to avoid confusion with a specific metric named "accuracy" defined previously in literature [27] and used by one of the techniques we studied [23].

the transferred test takes more steps than constructing the ground-truth test from scratch.

4.2 FrUITeR's Workflow

To address Req_2, Req_3, and partially Req_4 from Section 3, we designed an *automated* evaluation workflow with *customizable* components, shown in Figure 2. The goal of *FrUITeR*'s workflow is to generate *reproducible* evaluation results for a test-reuse technique's core functionality. The workflow's automation is enabled by two key aspects: (1) the *uniform representation* of the inputs and artifacts needed in the evaluation process, and (2) a set of customizable components that output the evaluation results of interest automatically.

4.2.1 ***Uniform Representation of Inputs.*** As Figure 2 shows, *FrUITeR* takes two types of input: Test Input (bottom-left) and Mapping Input (top-right). The two are a combination of inputs taken and artifacts produced by existing test-reuse techniques, as well as three new inputs introduced in *FrUITeR* to automate the evaluation process: Ground-Truth Tests, GUI Maps and Canonical Maps.

Test Input contains source tests, ground-truth tests, and transferred tests as defined in Section 2.1. The tests may be captured in various forms by a test-reuse technique, and we cannot assume that they will be analyzable in a standard way *a priori*. For instance, all tests in ATM [21] and GTM [20] are represented as Espresso tests [6] in Java, while CraftDroid [25]'s source tests are written in Python using Appium [2] and its transferred tests are represented in JSON [15]. In order to enable their automated evaluation, the heterogeneous tests that are part of *FrUITeR*'s Test Input thus need to be standardized. *FrUITeR*'s Event Extractor converts the Test Input into a standardized representation of source events, ground-truth events, and transferred events as detailed in Section 4.2.2.

Mapping Input consists of the GUI Map and the Canonical Map, which enable automated evaluation of a test-reuse technique's *fidelity*. The two maps are newly introduced by *FrUITeR* and captured using a standardized representation. The *GUI Map* contains the *GUI event mapping* from a source app to a target app generated by a given test-reuse technique, and is used to compute the fidelity metrics introduced in Section 4.1.1. Prior work does not provide GUI Maps, but only the final Transferred Tests. The events in these tests cannot be used to calculate fidelity by comparing with source events directly, because the transferred events may include *ancillary* and *null* events. We further illustrate how we extract the GUI Maps from existing techniques and evaluate their fidelity automatically with *FrUITeR* in Section 5. On the other hand, the *Canonical Map* contains

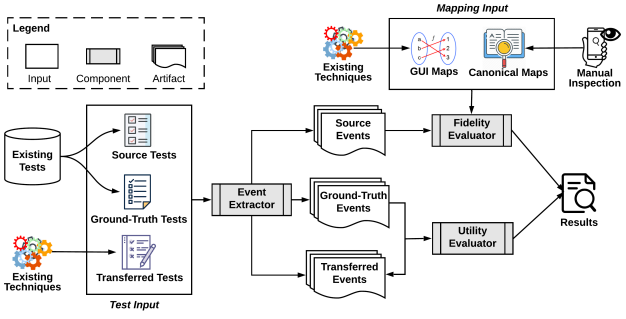


Figure 2: Overview of *FrUITeR*'s automated workflow.

the mapping from app-specific events to canonical events. This map is manually constructed and is used as the ground-truth mapping for *FrUITeR*'s Fidelity Evaluator component discussed below. Note that AppFlow [23] can generate a Canonical Map automatically using ML techniques. However, AppFlow's mapping results may be wrong, and thus cannot be used as the ground truth.

4.2.2 ***Customizable Components.*** *FrUITeR* introduces three customizable components, shown as shaded boxes in Figure 2: Event Extractor, Fidelity Evaluator, and Utility Evaluator.

Event Extractor leverages program analysis to extract the GUI event sequence from the usage tests' code. The sequence is represented as each event's ID or XPath, depending on which of the two is used in the test. ID and XPath are widely used to locate specific GUI elements in tests in various domains, including Android apps [7] and web apps [14]. For simplicity, we will use "ID" to refer to either the ID or XPath of a specific event in the rest of the paper.

To extract the event sequence, Event Extractor analyzes the Test Input to locate the program point of each event based on its corresponding API, e.g., click [4] or sendKeys [10] for tests written with Appium [2]. Once it identifies the location, Event Extractor determines the event's caller, i.e., the GUI element where the event is triggered, and performs a def-use analysis [19] to trace back the definition of the caller's ID. This definition is specified in a given API of the testing framework, such as findElementById() in Appium [18]. In that case, the def-use analysis is used to pinpoint the findElementById() call that corresponds to the event's caller so that ID's value can be determined. The input value associated with the event (if any) is determined by def-use analysis in the same manner. In the end, the converted Source Events, Ground-Truth Events, and Transferred Events are represented in a uniform way with IDs regardless of what testing framework is used.

Event Extractor is easily customizable to process tests written with different frameworks by replacing the relevant APIs' signatures. For instance, when identifying an event caller's ID, the relevant API is findElementById() if using Appium to test mobile apps, or findElement() if using Selenium to test web apps [16, 17].

Fidelity Evaluator takes the Source Events produced by Event Extractor and Mapping Input, and automatically outputs the sets of (1) *correct*, (2) *incorrect*, (3) *missed*, and (4) *nonExist* cases for calculating *FrUITeR*'s seven fidelity metrics (recall Table 1).

Algorithm 1 describes Fidelity Evaluator in detail. The algorithm iterates through each source event to determine to which of the four cases it should be assigned (Lines 2-16). To do so, it first gets the current source event (*src*), and the transferred event mapped from it (*trans*) based on the GUI Map (Lines 3-4). It then converts the app-specific events *src* and *trans* into their corresponding canonical events *srcCan* and *transCan*, using their respective Canonical Maps, so that the events are comparable (Lines 5-6). Finally, to determine which of the four cases *src* falls into, the algorithm first checks whether *trans* is a *null* event. If not, *transCan* will be compared against *srcCan* to determine whether the transferred event refers to the same canonical event as the source event, and *src* will be added to either the *correct* or *incorrect* set accordingly (Lines 7-11). If *trans* is *null*, the source event has not been mapped to any events in the target app. The algorithm then iterates through the Canonical Map of the target app (*tgtCanMap*) to determine whether the matching

Algorithm 1: FIDELITY EVALUATOR

```
   Input: EventList srcEvents, GUIMap guiMap,
          CanonicalMap srcCanMap, tgtCanMap
   Output: Sets correct, incorrect, missed, nonExist
1  correct = incorrect = missed = nonExist = ∅
2  for i = 1 to srcEvents.size do
3      src ← srcEvents.GET(i)
4      trans ← guiMap.GETMAPPED(src)
5      srcCan ← srcCanMap.GETCANONICAL(src)
6      transCan ← tgtCanMap.GETCANONICAL(trans)
7      if trans != null then
8          if transCan == srcCan then
9              correct.PUT(src)
10         else
11             incorrect.PUT(src)
12     else
13         if tgtCanMap.CONTAINS(srcCan) then
14             missed.PUT(src)
15         else
16             nonExist.PUT(src)
17 return correct, incorrect, missed, nonExist
```

event *srcCan* exists in the target app, and *src* will be added to either the *missed* set or *nonExist* set accordingly (Lines 12-16).

Utility Evaluator automatically analyzes the Ground-Truth Events and Transferred Events produced by Event Extractor. It uses this information to compute the two utility metrics—*effort* and *reduction*—based on their definitions described in Section 4.1.2.

4.2.3 ***Relationship to FrUITeR's Requirements.*** The workflow yields three benefits that target Req_2, Req_3, and Req_4 from Section 3.

First, the only manual effort required by *FrUITeR* is to construct the Canonical Maps by relating app-specific events to canonical events. This is a one-time effort per app, and each event only needs to be labeled once regardless of how many times it appears in a test (Req_2). By contrast, in previous work [20, 21, 25], each app-specific event needs to be manually labeled every time it appears in a test, possibly resulting in thousands of manual inspections.

Second, *FrUITeR* establishes ground truths with uniform representations: Canonical Maps are the ground truth for assessing *fidelity*, while Ground-Truth Events help to assess *utility*. This renders the evaluation results yielded by *FrUITeR* reproducible (Req_3). For instance, any mistakes or subjective judgments made in the current techniques' manual evaluation processes can be easily located by inspecting the Canonical Maps, and independently reproduced. Further, *FrUITeR*'s Canonical Maps are reusable, modifiable, and extensible for subsequent studies, helping to avoid duplicated work.

Third, *FrUITeR*'s workflow consists of customizable modules that isolate the evaluation to a relevant component of a test-reuse technique (Req_4). For instance, Fidelity Evaluator only assesses the performance of GUI event mapping, instead of evaluating a technique as a whole. Moreover, both Fidelity Evaluator and Utility Evaluator can be customized, reused, or extended to automatically evaluate other metrics of interest based on the standardized inputs and artifacts that *FrUITeR* defines, directly fostering future research.

4.3 *FrUITeR*'s Baseline Techniques

To better understand the performance of a test-reuse technique, we developed two baseline techniques—*Naïve* and *Perfect*—that establish the lower- and upper- bounds achievable by the fidelity and utility metrics in a given scenario.

Algorithm 2: NAÏVE BASELINE TECHNIQUE

```
   Input: EventList srcEvents, AppInfo tgtAppInfo
   Output: EventList transEvents
1  transEvents ← ∅
2  currentAct ← tgtAppInfo.GETMAINACTIVITY()
3  foreach src ∈ srcEvents do
4      isMapped ← FALSE
5      events ← tgtAppMap.GETALLEVENTS(currentAct)
6      events.RANDOMIZEORDER()
7      foreach event ∈ events do
8          if event.ACTION == src.ACTION then
9              similarity ← GETRANDOMSIMILARITY(0, 1)
10             if similarity > Threshold then
11                 transEvents.ADD(event)
12                 currentAct ← event.NEXTACTIVITY()
13                 isMapped ← TRUE
14                 break
15     if ¬isMapped then
16         transEvents.ADD(null)
17 return transEvents
```

4.3.1 ***Naïve Baseline.*** The Naïve baseline uses a random strategy to select the events in a target app to which each source event should be mapped. This sets the practical lower-bound of *fidelity*. As Algorithm 2 shows, Naïve initially explores the target app from the main Activity [9] (Line 2). For each source event, it obtains all the events at the current Activity (*events*) in a random order (Lines 5-6), and then tries to find a match between the current source event *src* and each event in *events* (Lines 7-14). When mapping *src* to *event*, Naïve first checks if the associated actions of the two events are the same, and only computes the similarity score when they are. The similarity score is computed by selecting a random value between 0 and 1 (Line 9), which are the lower and upper bounds used in existing work. If the similarity score of *src* and *event* is above a certain threshold, *event* is added to the list maintained in *transEvents* (Line 11). At that point, Naïve continues to explore the target app from the Activity reached by the transferred *event* (Line 12), and marks the current source event *src* as mapped (Line 13). In the end, if the source event is not mapped, it will be marked as a *null event* and added to *transEvents* (Line 15-16). Null events correspond to either the True Negative or True Positive categories in Table 1.

4.3.2 ***Perfect Baseline.*** The Perfect baseline transfers the source events based on the ground-truth mapping we establish (recall Section 4.2), assuming all source events are correctly mapped to the target app. We are particularly interested in the *utility* achieved by the Perfect baseline since it represents the upper-bound of the transferred tests' practical usefulness, which is not considered by existing work. This can help us identify the "room for improvement" and guide future research in test-reuse techniques.

5 FrUITeR's INSTANTIATION

This section describes how we instantiate *FrUITeR* to automatically evaluate the relevant individual modules of existing test-reuse techniques alongside *FrUITeR*'s baseline techniques, in partial satisfaction of Req_4 from Section 3. The evaluation is performed based on

FrUITeR's reusable benchmark, which addresses Req_5. To this end, we needed to provide information that enables *FrUITeR*'s automated workflow discussed in Section 4.2 and depicted in Figure 2: the Source Tests and Ground-Truth Tests, which are supplied as inputs to existing techniques; the Transferred Tests and GUI Maps, which are produced as outputs of existing techniques; and the manually constructed Canonical Maps. Section 5.1 explains how we mitigated the challenges faced in extracting the relevant components from existing techniques in order to generate their Transferred Tests and GUI Maps. Section 5.2 presents *FrUITeR*'s reusable, expandable benchmark for unbiased evaluation of test-reuse techniques, which contains the Source Tests, Ground-Truth Tests, and Canonical Maps used in *FrUITeR*'s automated workflow. Finally, Section 5.3 provides the details of *FrUITeR*'s implementation and generated data.

5.1 Modularizing Existing Techniques

To lay the foundation for addressing Req_4, we modularized *FrUITeR*'s design. In turn, this isolated the evaluation of GUI event mapping's fidelity and the transferred tests' utility, as discussed in Section 4.2. However, the existing techniques are implemented and evaluated as fully integrated, one-off solutions that do not provide the artifacts needed by *FrUITeR* to generate the modularized evaluation results. Because of this, we had to extract the specific functionality from existing techniques' implementations that performs the GUI event mapping (recall Section 2.2). Once the GUI Maps are available, we can generate the Transferred Tests used in *FrUITeR*'s Utility Evaluator. Note that the step of extracting GUI Mapper components is not needed for future test-reuse techniques if they follow *FrUITeR*'s modularized design. For example, we directly applied *FrUITeR* on the two baseline techniques we developed, with no extra effort.

Extracting the GUI Mapper components from the existing techniques was challenging since we had to understand each technique's design and implementation in detail, and to modify its source code. To this end, in addition to the available publications, we studied in depth the existing approaches' implementations [1, 3, 5, 8] and communicated with their authors extensively. We describe the challenges we faced during this process and the specific component-extraction strategies we applied to each existing solution.

5.1.1 ***Extracting AppFlow's GUI Mapper.*** AppFlow [23] is an ML-based technique whose key component trains a classifier that maps app-specific events to canonical events, but does not map the events from a source app to a target app. To compare AppFlow with similarity-based techniques, we leverage its Canonical Maps to transfer the source events to the target app by (1) mapping each source event to the corresponding canonical event based on the source app's Canonical Map and (2) mapping this canonical event back to the app-specific event in the target app based on the target app's Canonical Map. AppFlow's implementation does not output its Canonical Maps, so we had to locate and modify the relevant component to do so. Moreover, AppFlow does not store its trained classifier, so we had to configure its ML model and re-train it. During this process, we communicated with AppFlow's authors closely to understand its code, to obtain proper configuration files and training data, and to ensure the correctness of our re-implementation.

5.1.2 ***Extracting ATM's and GTM's GUI Mappers.*** As discussed earlier, ATM [21] was developed as an enhancement to GTM [20] and was shown to outperform it [21]. However, the authors of these two techniques compared them only on half of the source-target app pairs used in ATM's publication [21] due to the large manual effort required. Since *FrUITeR* largely automates the comparison process, we decided to extract the GUI Mapper components from both techniques to enable their comparison at a large scale.

An obstacle we had to overcome was that ATM and GTM both require the app's source code due to their use of the Espresso framework [6]. Thus, they cannot be compared as-is with techniques evaluated on closed-sourced apps, which would have limited our choice of benchmark apps. We discussed this issue with ATM's and GTM's authors and learned that the only step that requires source code for both techniques' GUI Mappers is computing the textual similarity score of image GUI elements (e.g., ImageButton). In that case, the text of the image's filename is retrieved from the app's code and analyzed to compute the similarity score. However, the main author confirmed that, in her experience, this feature is rarely needed in practice. We thus decided to extract ATM's and GTM's GUI Mapper components as stand-alone Java programs that do not require Espresso, omitting the filename-retrieval feature. We subsequently confirmed with the two techniques' authors the correctness of our implementation.

5.1.3 ***Extracting CraftDroid's GUI Mapper.*** CraftDroid's [25] implementation is only partially available. Its authors informed us that two of CraftDroid's modules—Test Augmentation and Model Extraction—were not releasable when we requested them, due to ongoing modifications, while the prior versions of the two modules were no longer available. The authors confirmed our observation that CraftDroid's GUI mapping functionality depends on the outputs of the two missing modules, and advised us that the best strategy would be for us to reimplement them based on CraftDroid's lone publication [25]. However, the publication in question is missing a number of details that would introduce bias in our re-implementation: we would have no guarantee that the versions of the two components we produce are the same as those used in CraftDroid. Instead, we decided to rely on CraftDroid's published Transferred Tests [5] in our evaluation.

To obtain CraftDroid's GUI Maps, we inspected its published artifacts [5] and found that only certain events in the Transferred Tests have associated similarity scores, while other events are labeled as "empty". Further investigation showed that each event in the Transferred Tests belongs to one of three cases: (1) events with available similarity scores are successfully mapped from the source events; (2) "empty" events are mapped from the source events but no match is found by CraftDroid (i.e., null events); (3) the remaining events are not mapped from the source events but are added by CraftDroid (i.e., ancillary events). We excluded the ancillary events so that the resulting transferred events have a 1-to-1 mapping from the source events, giving us CraftDroid's GUI Maps.

5.2 *FrUITeR*'s Benchmark

As discussed above in the motivation for Req_5, existing test-reuse techniques are evaluated on different apps and tests, which hinders their comparability. To address this, we established a reusable,

technique-independent benchmark. This section discusses our strategy for including existing apps and tests in the benchmark, and for generating the required ground truth.

5.2.1 ***Benchmark Apps and Tests.*** To maximize the results from existing work that we can attempt to reproduce, we first included the intersection of the subject apps used by existing work. This yielded 3 shopping apps: Geek, Wish, and Etsy. We further randomly selected 7 additional shopping apps and 10 news apps used by AppFlow [23]. This gave us 20 benchmark apps in total, as described in Table 2. Our rationale behind this choice of apps was two-fold: (1) AppFlow's authors manually inspected all app categories on Google Play and identified shopping and news as categories with common functionalities suitable for test reuse; (2) AppFlow was evaluated on the largest number of subject apps among the existing techniques. By comparison, ATM [21] used 16 open-source apps that are not as popular as those used in AppFlow.

To construct the benchmark tests, we further followed the test cases defined in AppFlow, with a similar rationale: (1) AppFlow's authors conducted an extensive study to manually identify tests that are shared in shopping and news apps; (2) AppFlow defines a larger number of tests compared to other work. For example, CraftDroid [25] only has 2 tests defined in each app category. We excluded those tests that require mocking external dependencies (e.g., a payment service). This resulted in 15 tests in the shopping category and 14 tests in the news category, shown in Table 3. Note that we cannot reuse AppFlow's tests directly because they are written in a special-purpose language defined by AppFlow for an entire app category rather than a specific app. Instead, we relied on multiple undergraduate and graduate students with Android experience to write the applicable tests for each of the 20 subject apps using Appium [2]. Some benchmark apps did not have each functionality described in Table 3, ultimately resulting in a total of 239 tests involving 1,082 events across the 20 apps (the two right-most columns of Table 2), requiring 3,920 SLOC of Java code.

5.2.2 ***Benchmark Ground Truth.*** As described in Section 4.2, we define Canonical Maps to represent the ground truth for the *fidelity* of the GUI event mapping, and Ground-Truth Events to represent the ground truth for the *utility* of the transferred tests.

Table 2: Summary information of benchmark apps.

	App ID	App Name	#Downloads	#Tests	#Events
Shopping	S1	AliExpress	100M	15	76
	S2	Ebay	100M	13	48
	S3	Etsy	10M	13	55
	S4	5miles	5M	12	78
	S5	Geek	10M	13	85
	S6	Google Shopping	1M	15	72
	S7	Groupon	50M	14	66
	S8	Home	10M	14	98
	S9	6PM	500K	14	63
	S10	Wish	100M	14	85
News	N1	The Guardian	5M	13	76
	N2	ABC News	5M	9	31
	N3	USA Today	5M	11	28
	N4	News Republic	50M	10	40
	N5	BuzzFeed	5M	11	50
	N6	Fox News	10M	11	28
	N7	SmartNews	10M	9	20
	N8	BBC News	10M	9	22
	N9	Reuters	1M	10	37
	N10	CNN	10M	9	24

In our benchmark, we define 72 canonical events for the shopping apps and 55 for the news apps. Our canonical events are extended from AppFlow, aiming to reflect a finer-grained classification of GUI events. For instance, event "password" in the *sign-in* test (TS1/TN1 in Table 3), and events "password" and "confirm password" in the *sign-up* test (TS2/TN2 in Table 3), are all represented as the same canonical event "Password" in AppFlow. However, it is debatable whether that is appropriate. For example, mapping "password" in *sign-up* to "password" in *sign-in* may lead to non-executable tests. To remove ambiguity, we capture such events separately.

Based on the canonical events, we construct 20 Canonical Maps, one per subject app. We do so by manually relating to the canonical events a total of 561 subject apps' GUI events that appear in one or more of the 239 tests. As discussed in Section 4.2, this is the only manual step required by *FrUITeR* and is a one-time effort: the Canonical Maps can be reused when relying on the same subject apps. As a point of comparison, recall from Section 3 that evaluating 48 app pairs in ATM [21] required manually inspecting 2,400 events. By contrast, our one-time inspection of the 561 events enabled the use of 200 app pairs (2 categories × 10×10 apps, i.e., including an app's test transfer to itself) by every technique *FrUITeR* evaluated.

The Ground-Truth Events in our benchmark are extracted from the 239 tests by *FrUITeR*'s Event Extractor (recall Figure 2).

5.3 FrUITeR's Implementation Artifacts

FrUITeR's artifacts are publicly available [12]: its source code; final datasets; GUI Mappers extracted from existing work; implementations of baseline techniques, their GUI Maps, and Transferred Tests; benchmark apps and tests; and manually constructed benchmark ground truths. We highlight the key details of these artifacts below.

5.3.1 ***Source Code.*** *FrUITeR*'s Event Extractor (recall Figure 2) is implemented in Java using Soot [11] (235 SLOC). *FrUITeR*'s Fidelity Evaluator and Utility Evaluator are implemented in Python (1,045 SLOC). *FrUITeR*'s baseline techniques Naïve and Perfect (recall Section 4.3) are likewise implemented in Python (112 SLOC). The GUI Mapper components extracted from existing techniques (recall Section 5.1) are implemented in their original programing languages: AppFlow in Python (1,084 SLOC); GTM in Java (1,409 SLOC); and ATM in Java (1,314 SLOC). The functionality that processes their

Table 3: Benchmark test cases in shopping (TS) and news (TN) categories.

Test ID	Test Case Name	Tested Functionalities
TS1/TN1	Sign In	provide username and password to sign in
TS2/TN2	Sign Up	provide required information to sign up
TS3/TN3	Search	use search bar to search a product/news
TS4/TN4	Detail	find and open details of the first search result item
TS5/TN5	Category	find first category and open browsing page for it
TS6/TN6	About	find and open about information of the app
TS7/TN7	Account	find and open account management page
TS8/TN8	Help	find and open help page of the app
TS9/TN9	Menu	find and open primary app menu
TS10/TN10	Contact	find and open contact page of the app
TS11/TN11	Terms	find and open legal information of the app
TS12	Add Cart	add the first search result item to cart
TS13	Remove Cart	open cart and remove the first item from cart
TS14	Address	add a new address to the account
TS15	Filter	filter/sort search results
TN12	Add Bookmark	add first search result item to the bookmark
TN13	Remove Bookmark	open the bookmark and remove first item from it
TN14	Textsize	change text size

outputs and generates the uniform representation of GUI Maps and Transferred Tests is implemented in Python (404 SLOC). As discussed earlier, due to CraftDroid's unavailable source code, we can only interpret its published artifacts [5]; that functionality is implemented in Python (86 SLOC). The data analyses that interpret our final datasets are written in R (585 SLOC).

5.3.2 ***Final Datasets.*** Our final datasets contain the results of 11,917 test transfer cases generated by the GUI Mappers from the existing techniques and our two baselines when applied on *FrUITeR*'s benchmark. We apply 5 techniques—AppFlow, ATM, GTM, Naïve, Perfect—to transfer tests across 20 shopping and news apps, involving 1,000 source-target app pairs. This yielded 2,381 result entries per technique. As discussed earlier, we have to rely on CraftDroid's final results, and can thus only compare CraftDroid to the other techniques on the 3 shopping apps—Geek, Wish, Etsy—used both in our benchmark and in CraftDroid's evaluation. This gave us only 12 result entries for CraftDroid. Each of the total 11,917 result entries contains the following information: (1) the source and target apps; (2) the source, transferred, and ground-truth tests; (3) the technique used to transfer the test; (4) the *correct*/*incorrect*/*missed*/*nonExist* sets of GUI events output by *FrUITeR*'s Fidelity Evaluator as described in Algorithm 1, and the seven corresponding fidelity metrics defined in Section 4.1.1; and (5) values of the two *utility* metrics—*effort* and *reduction*—defined in Section 4.1.2. Note that obtaining these 11,917 result entries following prior work's evaluation processes would have required manual inspection of 53,963 events that appear across all of the source tests, which is infeasible in practice.

6 FINDINGS

The datasets produced by *FrUITeR* include the results obtained by evaluating side-by-side the extracted key components from the four existing test-reuse techniques for Android apps and the two baseline techniques we developed. In turn, this data enables further in-depth studies of a range of research questions in this emerging domain. As an illustration, this section highlights several findings uncovered by *FrUITeR*'s datasets that are missed by prior work.

6.1 GUI Mapper Comparison

As discussed earlier, existing techniques are evaluated in their entirety, on different benchmark apps and tests, and using different evaluation metrics, all of which makes their results hard to compare. By contrast, *FrUITeR* was able to evaluate their extracted GUI Mappers side-by-side, with our two techniques—Naïve and Perfect—serving as baselines. We note that it is possible for a given test-reuse technique to produce results as a whole that may be different from those produced only by its extracted GUI Mapper. One reason may be that there is additional relevant functionality that is scattered across the technique's implementation, whose ultimate impact its authors may not have realized when we consulted them. However, any such functionality can be easily added to the existing GUI Mappers, or introduced in additional *FrUITeR* components.

6.1.1 ***Fidelity Comparison.*** *FrUITeR*'s datasets [12] contain the results of all seven fidelity metrics from Section 4.1.1 obtained using our benchmark. Due to space limitations, we restrict our discussion to *Precision* and *Recall*; the remaining metrics followed similar trends. Figure 3 shows the average precision and recall achieved by the four existing techniques as well as Naïve; we omit Perfect since its values are always 100% by definition.

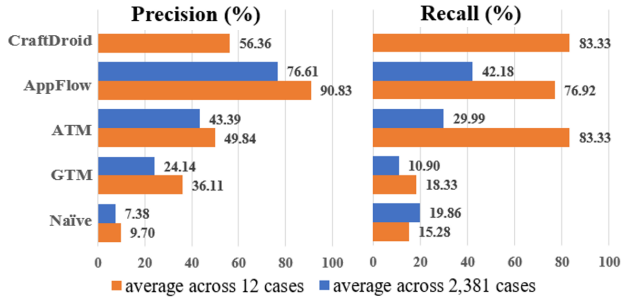


Figure 3: Comparison of average *precision* and *recall*.

For each technique except CraftDroid, the top (blue) bar shows the average calculated based on 2,381 cases transferred among both shopping and news apps. To meaningfully compare CraftDroid with other techniques, even if only partially, we show the averages calculated based on the 12 cases for which we have CraftDroid's data, in the bottom (orange) bars. CraftDroid only transferred "Sign In" and "Sign Up" tests in the 3 shopping apps—Geek, Wish, and Etsy—leading to the 12 cases (6 source-target app pairs × 2 tests).

We highlight three observations based on the results from Figure 3. First, every existing technique yields lower recall than precision on the larger (blue) data set, meaning that it suffers from more *missed* (i.e., false negative) than *incorrect* (i.e., false positive) cases. Although its recall is highest among the existing techniques, AppFlow exhibits the largest drop-off between its precision and recall values. A plausible explanation is that, as an ML-based technique, AppFlow will likely fail to recognize relevant GUI events if no similar events exist in its training data. This was somewhat unexpected, however, given that AppFlow's authors carefully crafted its ML model to the app categories we also used in *FrUITeR*, and suggests that additional research is needed in selecting and training effective ML models for UI test reuse. By comparison, similarity-based techniques such as ATM will miss fewer GUI events in principle: they can always compute a similarity score between two events and return the mapped events whose scores are above a given threshold. However, if the similarity threshold is set too low, it will result in more *incorrect* cases, leading to low precision.

A related observation is that AppFlow's precision outperforms the other techniques across the board, for both the larger (blue) and smaller (orange) datasets. This is because AppFlow has the advantage of more information, obtained from a large corpus of apps in its training dataset, than the similarity-based techniques, which compute the similarity scores based only on the information extracted from the source and target apps under analysis. However, AppFlow's recall is lower than both ATM and CraftDroid on the 12 (orange) cases from Geek, Wish, and Etsy. This reinforces the above observation that an ML-based technique will fail to recognize GUI events if no similar events exist in its training data.

Finally, our data confirms that ATM indeed improves upon GTM, as indicated in their pairwise comparisons across both precision and recall, and large and small datasets. In fact, GTM exhibits the lowest fidelity of all existing techniques, and its recall across the 2,381 (blue) cases is actually lower than that achieved by the Naïve

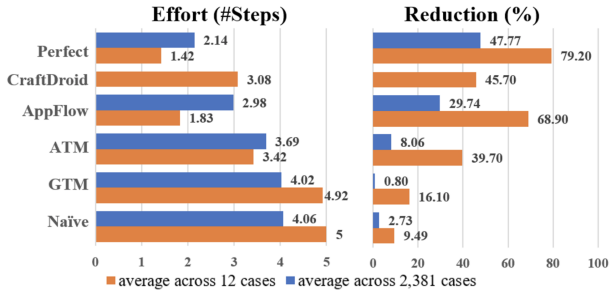


Figure 4: Comparison of average *effort* and *reduction*.

strategy. We note that GTM's design is geared to transferring tests in programming assignments that share identical requirements, and is clearly not suited to heterogenous real-world apps.

6.1.2 ***Utility Comparison.*** Figure 4 shows the two utility metrics yielded by each of the four existing and two baselines. Recall from Section 4.1.2 that utility measures how useful the transferred tests are in practice compared to the ground-truth tests. The objective of utility is to minimize the *effort* while maximizing the *reduction*.

The utility of existing techniques shows similar trends to those observed in the case of fidelity. For example, AppFlow outperforms other techniques, while GTM exhibits similar performance to that of Naïve. This indicates a possible correlation between the fidelity of the GUI event mapping and the utility of the transferred tests.

At the same time, we observe that, while our Perfect GUI Mapper achieves higher utility than the remaining techniques, that utility is not optimal. In fact, Perfect's average *reduction* is under 50% across the 2,381 cases in the larger dataset (top, blue bar). In other words, even with the best possible mapping strategy, we save less than half of the effort required to complete the task manually. The previously published techniques perform much worse than this: AppFlow saves under 30%, ATM under 10%, and GTM under 1% of the required manual effort, while the reduction yielded by CraftDroid on the smaller (orange) dataset is lower than Perfect's on either of the two datasets. This indicates that fidelity is clearly not the only factor to consider in order to achieve desired utility, and that there is large room for improvement in future test reuse techniques.

To verify the above insights, we conducted pairwise correlation tests between the seven accuracy and two utility metrics. Overall, the results, further discussed below and provided in their entirety in *FrUITeR*'s online repository [12], show a *weak correlation* between fidelity and utility. This reinforces our observation that accurate GUI mappings can yield useful transferred tests, but are not the only relevant factor. In turn, this finding calls for exploration of other components in test-reuse techniques since the focus on GUI event mapping alone can hit a "ceiling", as shown by the Perfect baseline. We explore such possible directions next.

6.2 Insights and Future Directions

Guided by the above observations, we explore potential strategies for improving UI test reuse with various statistical tests and manual inspections on *FrUITeR*'s datasets. Due to space limitations, we highlight four findings that were not reported by previous work.

Source app selection matters for a given target app. Figures 3 and 4 both show consistent improvement across the techniques in the smaller data sets (12 cases transferred among 3 apps) compared to the larger ones (2,381 cases transferred among 20 apps). This suggests that certain source-target app pairs achieve better results than others. For example, we found that app pairs involving Wish, Geek, and a benchmark app called Home—all of which are developed by the same company, *Wish Inc.*—achieve high fidelity and utility, regardless of the technique used. Another such compatible app pair is ABC News and Reuters. Performing a large-scale evaluations enabled by *FrUITeR* will help spot pairings like this, and give researchers a starting point to explore the characteristics that can lead to better transfer results.

Automated transfer is not suitable for all tests. Our utility metrics revealed large *effort* and negative *reduction* in some cases, meaning that correcting a transferred test required more work than writing it from scratch. Further inspection revealed that this is primarily due to a test's length rather than a technique's accuracy. For instance, Perfect showed no benefit ($reduction \leq 0$) 16% of the time, and the average number of source events in those cases is only 4. This suggests that, for simple tests, manual construction may be preferable. Future research should consider the criteria for suitable tests to transfer instead of transferring all source tests.

There is a trade-off between ML- and similarity-based techniques. As discussed above, an insufficient *training set* in an ML-based technique may yield low recall, while a low *similarity threshold* in a similarity-based technique can address this but may yield low precision. This suggests two future research directions. First, selecting training sets and similarity thresholds is important, but existing techniques did not justify their choices [20, 21, 23, 25]. There is clearly a need for further study of novel strategies such as incorporating dynamic selection criteria based on target app characteristics. Second, future research should consider the trade-offs across different test-reuse techniques and provide guidance on selecting the most suitable techniques for a given scenario.

Test length is not a key factor influencing fidelity. CraftDroid and GTM studied the relationship between the test length and their transferred results. For instance, CraftDroid showed a strong negative correlation between test length and its two fidelity metrics (coefficient < -0.5 in both cases). To verify these findings, we conducted correlation tests on *FrUITeR*'s much larger datasets. Our results indicate a negative but very weak correlation between test length and *FrUITeR*'s fidelity metrics ($-0.25 <$ coefficient < 0 across all seven cases). This shows that test length is not the key factor that impacts fidelity, arguing that future research targeting reuse of complex tests may be a fruitful direction.

7 CONCLUSION

This paper has presented *FrUITeR*, a customizable framework for automatically evaluating UI test-reuse techniques. *FrUITeR* has been instantiated and successfully demonstrated on the key functionality extracted from existing test-reuse techniques that target Android apps. In the process, we have been able to identify several avenues of future research that prior work has either missed or actually flagged as not viable. We publicly release *FrUITeR*, its accompanying artifacts, and all of our evaluation data, as a way of fostering future research in this area of growing interest and importance.

REFERENCES

[1] 2019. AppFlow's source code and artifacts. https://github.com/columbia/appflow.
[2] 2019. Appium: Mobile App Automation Made Awesome. http://appium.io
[3] 2019. ATM's source code and artifacts. https://sites.google.com/view/apptestmigrator/.
[4] 2019. Click - Appium. http://appium.io/docs/en/commands/element/actions/click
[5] 2019. CraftDroid's source code and artifacts. https://sites.google.com/view/craftdroid/.
[6] 2019. Espresso. https://developer.android.com/training/testing/espresso.
[7] 2019. Find Elements - Appium. http://appium.io/docs/en/commands/element/find-elements
[8] 2019. GTM's source code and artifacts. https://sites.google.com/view/testmigration/.
[9] 2019. Introduction to Activities | Android Developers. https://developer.android.com/guide/components/activities/intro-activities
[10] 2019. Send Keys - Appium. http://appium.io/docs/en/commands/element/actions/send-keys
[11] 2019. Soot - A Java optimization framework. https://github.com/Sable/soot.
[12] 2020. FrUITeR's website with supplementary materials. https://anonymousresearchers.github.io/fruiter-website/.
[13] 2020. Gherkin Syntax - Cucumber Documentation. https://cucumber.io/docs/gherkin
[14] 2020. How to locate an element on the page - Web Performance. https://www.webperformance.com/load-testing-tools/blog/articles/real-browser-manual/building-a-testcase/how-locate-element-the-page
[15] 2020. JSON - Wikipedia. https://en.wikipedia.org/wiki/JSON
[16] 2020. SeleniumHQ Browser Automation. https://www.selenium.dev
[17] 2020. Web element :: Documentation for Selenium. https://selenium.dev/documentation/en/webdriver/web_element
[18] admin. 2019. Chapter-4: Appium Locator Finding Strategies - Kobiton. *Kobiton* (Apr 2019). https://kobiton.com/book/chapter-4-appium-locator-finding-strategies
[19] Frances E. Allen and John Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (1976), 137.
[20] Farnaz Behrang and Alessandro Orso. 2018. 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*.
[21] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *34th International Conference on Automated Software Engineering (ASE 2019)*.
[22] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet? (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[23] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 269–282.
[24] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.
[25] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *34th International Conference on Automated Software Engineering (ASE 2019)*.
[26] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.
[27] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval*. Cambridge university press.
[28] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Transferring Tests Across Web Applications. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 50–64.