AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests

Gang Hu
ganghu@cs.columbia.edu
Columbia University
New York, NY, United States

Linjie Zhu
linjie@cs.columbia.edu
Columbia University
New York, NY, United States

Junfeng Yang
junfeng@cs.columbia.edu
Columbia University
New York, NY, United States

ABSTRACT

UI testing is known to be difficult, especially as today's development cycles become faster. Manual UI testing is tedious, costly and error-prone. Automated UI tests are costly to write and maintain.

This paper presents AppFlow, a system for synthesizing highly robust, highly reusable UI tests. It leverages machine learning to automatically recognize common screens and widgets, relieving developers from writing ad hoc, fragile logic to use them in tests. It enables developers to write a library of modular tests for the main functionality of an app category (*e.g.*, an "add to cart" test for shopping apps). It can then quickly test a new app in the same category by synthesizing full tests from the modular ones in the library. By focusing on the main functionality, AppFlow provides "smoke testing" requiring little manual work. Optionally, developers can customize AppFlow by adding app-specific tests for completeness.

We evaluated AppFlow on 60 popular apps in the shopping and the news category, two case studies on the BBC news app and the JackThreads shopping app, and a user-study of 15 subjects on the Wish shopping app. Results show that AppFlow accurately recognizes screens and widgets, synthesizes highly robust and reusable tests, covers 46.6% of all automatable tests for Jackthreads with the tests it synthesizes, and reduces the effort to test a new app by up to 90%. Interestingly, it found eight bugs in the evaluated apps, including seven functionality bugs, despite that they were publicly released and supposedly went through thorough testing.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Empirical software validation*; Software evolution;

KEYWORDS

mobile testing; test reuse; test synthesis; UI testing; machine learning; UI recognition

ACM Reference Format:
Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3236024.3236055

1 INTRODUCTION

Most applications are designed to interact with humans, making it crucial to test the functionality, performance, and other key aspects of their user interfaces (UIs). Yet, UI testing is known to be exceedingly challenging. Manual UI testing has the advantage of testing faithful human experience, but the downside is that it is tedious, costly, and error prone – imagine a poor tester repeating 30 manual tests on 50 different devices. Automated testing is supposed to be the rescue, but today's test automation in industry requires a tremendous amount of developer "babysitting," and few companies have the skills or resources to set it up, as illustrated in the following comment at HackerNews [64], a top developer forum: "*I have worked in several companies that have had goals of automated UI regression test suites, but I've never worked at a company that pulled it off successfully.*"

UI Test automation often rely on script-based testing. Specially, to automate UI testing, developers must invest a high initial cost to write test scripts, diagnose test failures which are often caused by "broken" tests instead of bugs in the application code [69], and maintain test scripts when the application's UI evolves. While these tasks seem easy on the surface, numerous pitfalls make them daunting because application UIs are designed for human intelligence but test scripts are low-level, click-by-click scripts. For instance, while we humans can easily recognize without ambiguity the button to add an item to a shopping cart, whether or not the button shows "Add", "+", or an icon, a test script locates the button typically via a developer-provided, hardcoded method (*e.g.*, searching the internal widget ID or by text match). This hardcoded method can easily become incorrect when for example the button evolves from 'Add' to an icon or the application has different designs based on device factors such as screen size. Test record and replay [21, 23, 27, 33, 37, 70, 75] reduces the cost of writing tests, but recorded tests rarely work out of the box, and UI evolution still requires re-recording [45].

These test automation challenges are exacerbated by today's ever faster development cycles. Development trends such as Continuous Integration [40] and DevOps [39] require running tests on each code commit or merge to a key branch which may happen a dozen time a day, calling for fast, fully automated testing.

A standard software engineering practice to writing difficult code is to delegate: experts implement the code as a library or service, and other developers reuse. Examples include cryptography [13, 18], distributed consensus [19, 66, 67], and image processing [86]. In UI testing, there is ample opportunity for reusing tests because many apps are in the same category and implement similar user flows. For instance, almost all shopping apps implement some forms of user sign in, search for an item, check item details, add to shopping cart, check out, *etc.* We studied the top 309 non-game mobile apps and found that 15 app categories are enough to cover 196 or 63.4% apps (7.1), demonstrating the huge potential of sharing tests across apps in the same category. Thus, it would save much struggling if we could create a robust, reusable test library for shopping apps.

Unfortunately, few of today's automation frameworks are designed for reusing test scripts across apps. First, despite that apps in the same category share much similarity in their flows, they may have very different designs, texts, and names for their screens and widgets. Thus, a test script for an app often cannot locate the right screens and widgets for another app. Second, apps in the same category may still have subtly different flows. For instance, the sign-in flow of an app may contain just the sign-in screen, but another app may show a welcome screen first. The add-to-shopping-cart flow of an app may require a user to first visit the item details screen, but another app may allow users to add items in search results directly to shopping cart. These subtle differences prevent directly reusing test scripts on different apps.

This paper presents AppFlow, a system for synthesizing highly robust, highly reusable UI tests. It enables developers – *e.g.*, those in the "shopping app" community or a testing services company – to write a library of modular UI tests for the main functionality of a given category of apps. This library may be shared open-source or stored within a testing cloud service such as Google's Firebase Test Lab or Amazon's Device Farm. Then, when developers want to test a new app in the same category, they can quickly synthesize full tests from the modular ones in the library with a few lines of customization, greatly boosting productivity.

By focusing on the main functionality of an app category, AppFlow provides "smoke tests" or build verification testing for each source code change, requiring little or no manual work. Previous work [57] has shown that such tests, even incomplete, provide quick feedback to developers and help them fix bugs early before the bugs cause greater impact. Optionally, developers can customize AppFlow to add app-specific tests or override defaults to perform complete regression testing.

A key idea in AppFlow is a machine learning approach to recognizing screens and widgets. Instead of relying on developers' hard-coded logic, AppFlow learns a classifier from a training dataset of screens and widgets labeled with their intents, using a careful selection of features including texts, widget sizes, image recognition results of graphical icons, and optical character recognition (OCR) results. The training dataset can come from a developer community for an app category, and AppFlow provides several utilities to simplify this mostly one-time data collection. After the classifier is trained, AppFlow uses it to map variant screens and widgets to canonical ones. For instance, it maps text edit boxes with "Username", "Your Email", or "example@email.com" on sign-in screens all to `signin.username`, representing the user-name widget.

This machine learning approach enables the AppFlow tests to refer to *canonical* screens and widgets instead of app-specific ones, enjoying a variety of benefits. First, apps' UI can now evolve without breaking tests as long as the new designs can be recognized by AppFlow. Second, app UI can now respond to device factors such as screen size without breaking tests. Third, canonical screens and widgets abstract app-specific variations, making it easy to share tests across apps. Fourth, AppFlow's ability to recognize screens enables developers to focus on testing the specific flows of a screen without writing much boilerplate code to first bring the app to the screen or later restore the app to a previous state. This benefit is crucial for reusability, which we elaborate next.

A second key idea in AppFlow is to automatically discover apps' behaviors by applying reusable, self-contained tests called *flows* and synthesize full tests from them. To test a feature such as "at the item details page, a user can add the item to shopping cart", the developer writes a flow that contains three components: (1) the precondition of the test such as "app must be at item details screen;" (2) the postcondition of the test such as "app must be at shopping cart screen;" and (3) the actual steps to carry out the test such as "click Add button." The precondition and postcondition are in spirit similar to Hoare Logic, and can contain custom conditions on app state such as `loggedin = true` (*i.e.*, the user must have logged in). This flow is dual-purpose: it can be used to test if an app implements this feature correctly, and it can be used to navigate an app into states which are required to test other features. Specifically, given a library of flows, AppFlow dynamically synthesizes full tests as follows: it starts the app, recognizes its state, finds activated flows whose preconditions are met, executes each flow, and repeats for each new state reached.

AppFlow's synthesis has two main benefits. First, it greatly simplifies test creation because developers no longer need to write boilerplate code to bring the app to a certain state or clean up the state after. Second, modularization enables test reuse. If tests are specified as a whole, a test can hardly be reused due to variations of implementations of not only the scenario under test, but also the steps required to reach the scenario. In contrast, modular tests can be properly synthesized to adapt to a specific app's behavior. For instance, we can create a test library that contains two sign-in flows with or without the welcome screen and two add-to-shopping-cart flows passing or not passing item details screen. AppFlow can then synthesize the right tests for a new shopping app we want to test, mixing-and-matching the modular flows. In addition, it also allows AppFlow to adapt to apps' behavior changes. AppFlow can discover an app's new behaviors and automatically synthesize corresponding tests for them.

We implemented AppFlow for the Android platform because of its wide adoption and tough market competitions developers face, but the ideas and techniques are readily applicable to general UI testing. AppFlow's language to write flows is an extension of Gherkin [50], a human-readable domain-specific language for describing app behaviors.

Our evaluation of AppFlow consists of four sets of experiments. First, we evaluated AppFlow on 40 popular shopping apps and 20 news app by creating and reusing test libraries for the two app categories. Second, we conducted a case study of the BBC news app with two dramatically different versions to see if the tests AppFlow synthesizes are robust against the changes. Third, we conducted a user study of 15 subjects on creating tests for the Wish shopping app to compare AppFlow's approach vs writing tests using an existing test framework. Fourth, we analyzed a complete manual test plan from the developers of the JackThreads app and quantified how many tests AppFlow can automatically synthesize. Results show that AppFlow accurately recognizes screens and widgets, synthesizes highly robust and reusable tests, covers 46.6% of all automatable tests for Jackthreads, and reduces the effort to test a new app by up to 90%. Interestingly, it also found eight bugs in the evaluated apps, including seven functionality bugs, despite that they were already publicly released and supposedly went through thorough testing.

This paper makes three main contributions: (1) the AppFlow system for synthesizing highly robust, highly reusable tests; (2) our technique that leverages machine learning to recognize screens and widgets for robustness and reusability; and (3) our evaluation on 60 real-world shopping and news apps that produces 2944 tests and found 8 bugs. AppFlow's source code and the test libraries evaluated are available at github.com/columbia/appflow; AppFlow's dataset is available at github.com/columbia/appflow-dataset.

This paper is organized as follows. An overview of AppFlow is given in section 2. How machine learning is used is shown in section 3. Method to define flows is presented in section 4. The synthesis process is illustrated in section 5. Implementation details are discussed in section 6. We evaluated AppFlow in section 7. We discussed limitations of this approach in section 8. Related works are reviewed in section 9. We conclude in section 10.

2 OVERVIEW

This section first presents a succinct example to show how to write AppFlow tests (§2.1), and then describes its workflow (§2.2).

2.1 Example

```
Scenario: add to shopping cart [stay at cart]
  Given screen is detail
  And cart_filled is false
  When click @addtocart
  And click @cart
  And not see @empty_cart_msg
  Then screen is cart
  And set cart_filled to true
```

Figure 1: Flow: "add to shopping cart".

Suppose a developer wants to test the flow "adding an item to an empty shopping cart clears the 'shopping cart is empty' message" for shopping apps. Figure 1 shows an example for this test in AppFlow. "Given..." specifies the *precondition* of the flow. The screen to activate this flow should be the "detail" screen, the *canonical screen* that shows an item's details. This screen exists in almost all shopping apps, so using it to specify the condition not only eases the understanding of this flow, but also allows this flow to be reusable on other shopping apps. Here "screen" is a *visible property* built into AppFlow. In contrast, the flow specifies in the precondition that "cart_filled" must be "false," and "cart_filled" is a developer-defined *abstract property* indicating whether the shopping cart is filled. Abstract properties are intended to keep track of the invisible portions of app states, which can often be crucial for writing robust tests. To run this flow, AppFlow ensures that the precondition of the flow must be met, *i.e.*, all properties specified in the precondition must have the corresponding values.

Next, the flow does two clicks to the `@addtocart` and `@cart` buttons. Unlike traditional test scripts that refer to the widgets using handwritten, fragile logic, AppFlow tests use *canonical widgets* exported by a test library, and AppFlow leverages machine learning to match real widgets to canonical ones.

Then, the flow performs a check ("not see..."). After the two clicks, current screen must be the canonical screen "cart", which represents the shopping cart screen. Thus, the flow checks to ensure that the canonical widget `@empty_cart_msg`, which signals that the shopping cart is empty, should not be seen on the screen.

Finally, "Then" specifies in the *postcondition* that the screen after executing the clicks must be the canonical "cart" screen, which AppFlow will check after executing this flow. (Postconditions are different from checks because postconditions cause AppFlow to update the app state it maintains.) The flow also sets "cart_filled" to be "true" after executing this flow, which causes AppFlow to update the abstract properties it tracks to reflect this effect. After executing this flow, AppFlow will check to see if the new values of these properties satisfy the preconditions of any previously inactive flows, and add these flows to the set of flows to execute next.

This simple example shows some key benefits of AppFlow. This flow is easy to understand, even for non-developers (*e.g.*, a product manager). The canonical screens and widgets used are recognized by AppFlow automatically using machine learning methods, making the test robust against design changes and reusable across different apps. The system allows developers to describe just the flows to test without writing boilerplate code to bring the app to an item details screen.

2.2 Workflow

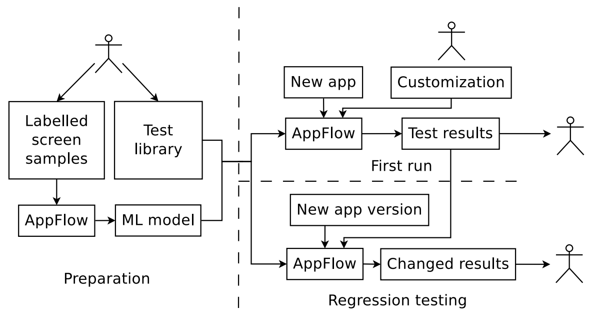


Figure 2: Workflow of AppFlow. The stick figure here represents developer intervention.

Figure 2 shows the workflow of AppFlow. It operates in two phases: the first phase, mostly one-time, prepares AppFlow for testing a new category of apps (§2.2.1), and the second phase applies AppFlow to test each new app in the category (§2.2.2).

2.2.1 Prepare for a New App Category. To prepare AppFlow for a new category of apps, developers do two things. First, they create a test library in AppFlow's language (§4) that contains common flows for this category, and define canonical screens and widgets during this process. Second, they use simple AppFlow utilities to capture a dataset of canonical screens and widgets and label them. Sometimes apps in different categories share similar screens (*e.g.*, sign-in screens), and these samples from other app categories can also be added. Given this dataset, AppFlow extracts key features from each sample and learns classifiers to recognize screens and widgets based on them (§3).

2.2.2 Test a New App. To test a new app for the first time, developers do two things. First, they customize the test library for their app. Machine learning is highly statistical and cannot always recognize every canonical screen and widget. To correct its occasional errors, developers run an interactive GUI utility of AppFlow to discover the machine learning errors and override them. In addition, developers supply values to the variables used in the library, such as the test user name and password. Developers may also add custom flows to test app-specific behaviors. The syntax and usage of this customization are described in §5.1.

Second, developers run AppFlow on the app to record the initial test results. Recall that a test library typically contains several variant flows such as signing in from the welcome screen or the menu screen. AppFlow runs all flows and reports the result for each, letting developers confirm which flows should succeed and which should fail.

Under the hood, AppFlow uses the flows in the test library to synthesize full tests through a systematic discovery process. Recall that a flow is active if its precondition is met in a state. At first, only the "start app" flow is active. In the discovery process, new app states and new paths to reach them are discovered, and more flows are activated. The process terminates when no more flows need to be tested. The detail of this process is explained in §5.2.

After the two setup steps, developers can now test new versions of the app regularly for regressions. AppFlow runs a similar process to synthesize full tests for each new app version, comparing the results to those from the previous run. It reports any unexpected failures and unexpected successes of the flows to developers, who should either fix any regressions or confirm intended changes to AppFlow.

3 RECOGNIZING CANONICAL SCREENS AND WIDGETS

Intuitively, screens and widgets for similar purposes should have similar appearance for good user experience, and similar names for ease of maintenance. However, simple rules cannot recognize them correctly, because of variations across apps and evolution of the same app over time. For example, the "login" button on the "sign in" screen may contain "Login", "Sign in", "Let me in", or even an icon showing an arrow. The underlying UI object usually has a class name of "Button", but sometimes it can be changed to "TextView" or even "RelativeLayout". Instead of using ad hoc, manually written rules to recognize widgets, AppFlow leverages machine learning to combine information from many available sources, thus it is much more robust.

Feature selection is key to accurate recognition, and it absorbed much of our effort. We experimented with a variety of feature combinations, settled with the following method. For each UI object (screen or widget), the features include its key attributes such as description text, size, whether it is clickable; the UI layout of the object; and the graphics. All features are converted to values between 0 and 1 in the final feature vector. Numerical features such as size are normalized using the maximum value. Boolean features such as whether a widget is clickable is converted to 0 or 1 directly. UI layout is converted to text via a pre-order tree traversal. Graphical features are handled in two ways. Button icons carry specific meanings, so they are converted to feature vectors by calculating their histogram of oriented gradients (HOG) [14]. Other graphical features are converted to text via OCR. All textual features including those converted from UI layouts and graphics are converted using Term Frequency–Inverse Document Frequency (TF-IDF). Intuitively, TF-IDF gives a higher weight if a term occurs in fewer documents (thus more descriminative) and more times in a document. Sometimes 2-gram is used to form terms from words in text. We show the effects of different feature selection schemes on accuracy in §7.2.

Besides feature selection schemes, we also experimented with different learning algorithms, and found that screen recognition and widget recognition need different algorithms. The following subsections describes the feature selection scheme and learning algorithm that yield the best accuracy for recognizing screens and widgets.

3.1 Classifying Screens

AppFlow uses three types of features to recognize canonical screens.

Screen layout The screen layout is a tree containing all the widgets on the screen. Different screens may have different numbers of widgets and feature vectors have to be of fixed length, so AppFlow converts the entire screen's UI layout to one text string. It traverses the tree in pre-order and, for each widget visited, it selects the text, identifier, the underlying UI object's class name, and other key attributes of the widget. For size, position and other non-text attributes, AppFlow generates a set of words to describe them. For instance, consider a search box widget. It is typically at the top of a search screen with a large width and small height. Its identifier typically contains "Search" and "Edit" to indicate that it is editable, and for implementing the search functionality. Given this search widget, AppFlow first generates a set of words describing the geometry of widget ("TOP" and "WIDE") and another set containing the word split of the identifier ("Search" and "Edit") using a rule-based algorithm. It then uses the Cartesian product of the two sets of words as the description of this widget. This Cartesian product works better than individual words because it captures the correlation between the geometry and identifier for recognizing widgets (*e.g.*, "TOPsearch" is very indicative of a search widget); it also works better than a concatenation of all words because it is more invariant to minor design differences (*e.g.*, with concatenation "TOPWIDESearch" and "TOPSearch" become different terms).

Screen snapshot A user understands a screen mostly based on the screen snapshot. To utilize this information, AppFlow performs OCR on the snapshot to extract texts inside it.

Class information AppFlow includes the class name of the screen's underlying UI object in the features it selects. In Android, the class is always a subclass of Activity. Developers tend to name screen classes with human readable names to ease maintenance.

From the training data set, we train a neural network classifier [76] that takes the screen feature vectors as inputs and outputs the canonical screen. It has 1 hidden layer with 68 neurons, optimized with a stochastic gradient-based optimizer [43].

3.2 Classifying Widgets

For each widget in the tree of widgets captured from a screen, AppFlow selects the following features.

Widget's text The text attribute of the widget is used. This usually equals to the text shown on the widget. The text attribute of the widget is the most evident clue of what the widget represents, because usually users understand its usage through text. However, other features are still needed. In some cases, the widget shows an image instead of text. In other cases, text is embedded into the image, and the text attribute is empty.

Widget's context The widget's description, identifier and class name are used. The description and identifier of a widget are evidences of its functionality, especially for widgets which have empty text attributes. The description is provided for accessibility uses, while the identifier is used by developers. The class name provides some useful information, such as whether this is a button or a text box, but it can be inaccurate.

Widget's metadata The widget's size, position, and some other attributes are used. The widget's metadata, combined with other information, increases the accuracy of the machine learning results. For example, in almost all apps, the "password" widget on the "sign in" screen has its "isPassword" attribute set to true, which helps the machine learning algorithm distinguish it from the "email" widget.

Neighbour information Some widgets can be identified by observing their neighbours. For example, an empty editable text box with no ID or description may be hard to recognize, but users can understand its usage by observing its neighbour with a label containing text "Email:". AppFlow includes the left sibling of the current widget in the feature vector.

OCR result OCR result of the widget's image is used. Some widgets do not have ID, text, or description. For traditional frameworks, these widgets are especially hard to refer to, while we found them fairly common among apps. Some other widgets have only generic IDs, such as "toolbar_button". In these cases, AppFlow uses features which humans use to identify them. A user usually recognizes a widget either through its text, or its appearance. This feature captures the text part, while the next feature captures the graphical part.

Graphical features The image of the widget is used. Some widgets, such as icons, use graphical features to hint users its functionality. For example, in almost all apps, the search icon looks like a magnifier. AppFlow uses the HOG descriptor, widely used in single symbol recognition, to vectorize this feature.

Vectorized points from the train set are used to train linear support vector machine [7] (SVM) classifiers. Every linear SVM classifier recognizes one canonical widget. The penalty parameter C is set to 0.1. SVMs are used because it achieves high accuracy while requiring little resources. Because the number of widgets is much larger than the number of screens, efficiency must be taken into account. Canonical widgets from different screens are classified using different sets of classifiers. To classify a widget, it is vectorized as above, and fed into all the classifiers of its canonical screen. If the classifier with the highest confidence score is higher than the configurable threshold, its corresponding canonical widget is given as the result. Otherwise the result is "not a canonical widget".

4 WRITING TEST FLOWS

This section first describes the language extensions we made to Gherkin to support writing test flows (§4.1), then explains some specifics on creating a test library and best practices (§4.2).

4.1 Language to Write Flows

AppFlow's flow language follows Gherkin's syntax. Gherkin is a requirement description language used by Behavior-Driven Development [9] tool cucumber [49], which in turn is used by Calabash [91], a widely used automated testing framework for mobile apps. We thus chose to extend Gherkin instead of another language because mobile developers should already have some familiarity with it.

In AppFlow, each flow is written as a scenario in Gherkin where lines in the precondition are prefixed by `Given`, steps of the test are prefixed by `When`, and lines in the postcondition and effect are prefixed by `Then`. Unlike in Gherkin which use natural languages for the conditions and step, AppFlow uses visible and abstract properties. Calabash [91] extends Gherkin to also include conditions on the visible UI states, but it does not support abstract properties.

The actions in a flow are specified using a verb followed by its arguments. The verbs are common operations and checks, such as "see", "click", and "text". The arguments can be widgets or values. For widgets, either canonical ones or real ones can be used. Canonical ones are referenced with `@<canonical widget name>`. Real ones are found using *locators* similar to how Calabash locates widgets. Simple methods such as "id(arg)", "text(arg)" and "desc(arg)" find widgets by comparing their corresponding attributes with the argument "arg," while method "marked(arg)" matches any of those attributes. Here "arg" may be a constant or a configuration variable indicated using `@<variable name>`.

Below we show four examples of flows. The first flow tests that a user can log in with correct credentials:

```
Scenario: perform user login
  Given screen is signin
  And loggedin is false
  When text @username '@email'
  And text @password '@password'
  And click @login
  Then screen is not signin
  And set loggedin to true
```

The second flow tests that a logged-in user can enter shopping cart from the "main" screen:

```
Scenario: enter shopping cart [signed in]
  Given screen is main
  And loggedin is true
  When click @cart
  Then screen is cart
```

The third flow tests that the "shopping cart is empty" message is shown on the "cart" screen when the shopping cart is empty:

```
  Scenario: check that cart is empty
    Given screen is cart
    And cart_filled is false
    Then see @cart_empty_msg
```

The last flow, which requires the shopping cart to be non-empty, removes the item from the shopping cart, and expects to see the "shopping cart is empty" message:

```
Scenario: remove from cart [with remove button]
  Given screen is cart
  And cart_filled is true
  When click @item_remove
  And see @cart_empty_msg
  Then set cart_filled to false
```

4.2 Creating a Test Library

Today developers write similar test cases for different apps in the same category, doing much redundant work. By contributing to a test library combined with AppFlow's ability to recognize canonical screens and widgets, developers can share their work, resulting in greatly improved productivity.

There are two subtleties in writing flows for a library. First, developers need to decide how many flows to include in the test library. There is a trade-off between the cost of creating custom flows and the cost of creating customizations. With more flows, the test library is more likely to include rare app behaviors, so less custom flows are needed. On the other hand, more flows in the test library usually means more rare canonical widgets, which have fewer samples from apps. Thus, these widgets may have lower classification accuracy, and having them requires more time to customize. Second, the same functionality may be implemented slightly differently across apps. As aforementioned (§1), the add-to-shopping-cart flow of an app may require a user to first visit the item details screen, but another app may allow users to add items in search results directly to shopping cart. Although conceptually these flows are the same test of the add-to-shopping-cart functionality, they need to be implemented differently. Therefore AppFlow supports the notion of a test that can have several variant flows, and tracks the flow(s) that works when testing a new app (§5).

Best practices. From our experience creating test libraries for two app categories, we learned four best practices. They help us create simple, general, and effective test libraries. We discuss them below.

First, flows should be modular for better reusability. Developers should avoid writing a long flow that does many checks and keep pre/postconditions as simple as possible. Precondtions and postcondions are simple depictions of the app states. The concept of app states naturally exists in traditional tests; testers and developers sometimes describe them in comments or write checks for them. When writing preconditions and postconditions, it takes no more effort than writing checks for traditional methods. Rich functionalities do not directly translate into complicated design because mobile apps tend to have a minimalism design to focus on providing content to users without unnessary cognitive load [4]. An app with rich functionalities usually has properties separated into fairly independent groups, and thus have simple preconditions and postconditions. Short flows with well-defined pre/postconditions are simple to write, easy to understand, and more likely to be reusable. For instance, most flows should not cross multiple screens. Instead, a flow should specify the screen where it can start executing and the screen it expects when its execution finishes, and it should not cross other screens during its execution.

Second, test flows should refer only to canonical screens and widgets. If a flow wants to check for a specific widget on the current screen, this widget should be defined as a canonical widget, then the test flow can refer it. Similarly, if the flow wants to verify a screen is the expected screen, the screen should be defined as a canonical screen. This practice avoids checks which leads to fragile flows, such as searching for specific strings on the screen to verify the screen or comparing widgets' text to find a specific widget.

Third, flows of common functionalities implemented by most apps should be included, while rare flows should be excluded from the test library. From our experience, it is crucial for classification results to be accurate. If there are misclassifications, developers would be confused by incorrect test results. Time spent by developers in debugging tests would likely be longer than time required to write a few custom flows. In addition, larger test library increases the exeuction time of AppFlow.

Forth, test flows should be kept simple. Complex flows are hard to generalize to other apps. As we mentioned above, it would be helpful in this respect if flows are splitted into smaller pieces and made modular. Also, the properties used in flows' conditions should also be kept at minimum, since having more properties increases the testing time by creating more combinations.

5 APPLYING A TEST LIBRARY TO A NEW APP

A developer applies a test library to her app in two stages. First, in the setup stage, when applying the library to her app for the first time, she configures and customizes the test library, specifically assigning necessary values to test variables such as test account name and overriding classification errors of machine learning. The developer may also add custom flows in this stage to test app-specific behaviors. Afterwards, she runs AppFlow to synthesize tests and record the pass and fail results. Note that a failed flow does not necessarily indicate an error. Recall that the same functionality may be implemented differently, so a failed flow may simply mean that it does not apply to the tested app.

Second, in the incremental stage, she applies the library to test a new version of the app. Specifically, AppFlow runs all tests synthesized for the previous version on the new version, retries all flows failed previously, and compares the results with the previous results. The differences may show that some previously passing flows fail now and other previously failing flows pass now. The developer can then fix errors or confirm that certain changes are intended. She may further customize the library if needed. Each incremental run takes much less time than the setup stage because AppFlow memorizes tests synthesized for the previous version.

Both stages are powered by the same AppFlow's automated test synthesis process to discover applicable flows and synthesize full tests. AppFlow starts from the initial state of an app, repeatedly executes active flows, and extends a state transition graph with new states reached by these flows. When there are no more active flows, the process is finished. A full test for a flow is synthesized by combining a chain of flows which starts at the initial state and ends at the flow.

In the remaining of this section, we describe how a developer customizes a test library (§5.1) and how AppFlow applies the test library with customizations on an app to synthesize full tests (§5.2).

5.1 Configuration and Customization

Developers customize a test library to an app in four steps. The first three steps are typically required only in the first run. First, developers assign values to test variables. A new app needs new values for these variables, because they contain app-specific test data, such as the user name and password to be used for login, the search keyword, etc. This data is straightforward to provide, and AppFlow also provides reasonable defaults for most of them, but developers can override them if they want. Developers may optionally change AppFlow's options to better suit their needs. Here is an example of this part:

```
email user@example.com
password verysecurepassword
```

Second, developers create matchers for screens and widgets to override machine learning errors. Although machine learning greatly reduces the need for developer-written screen and widget matchers, it inherently misclassifies in rare occasions, which developers must override. To ease the task, we build a GUI tool that helps developers inspect the machine learning results on their app and generate matchers if needed. A screenshot of this tool is shown in Figure 3. Operationally, the tool guides developers to navigate to their app's canonical screens defined in the test library, and overlays the recognition results on the app screen. When the developers find any classification error, they can easily generate a matcher to override the error. We discuss typical classification errors and how developers can fix them below.

A widget is misclassified in two ways. First, a canonical widget can be misclassified as a non-canonical widget. Developers can fix this by creating a widget matcher to help AppFlow recognize this widget. They first select the misclassified canonical widget, press the space key, and click or type the correct label in a pop-up dialog. The tool will generate a boilerplate matcher using the widget's properties. If its ID is unique within the screen, the generated matcher finds a widget with this ID. Otherwise, the tool will examine the widget's class, text, and description. If this widget's properties are not unique enough to generate the matcher, widgets containing it would also be examined. Second, a non-canonical widget can be classified as a non-existing canonical widget. Developers can fix this in a similar way to the first case. The only difference is that the label typed in should be empty. The tool will generate a "negative" matcher, which means that there is no such canonical widget on the current screen.

A screen is also misclassified in two ways. First, a canonical screen can be classified as another canonical screen. Developers can create a screen matcher to fix this. They press the "x" key to enter the screen matcher generating mode, click unique widgets which

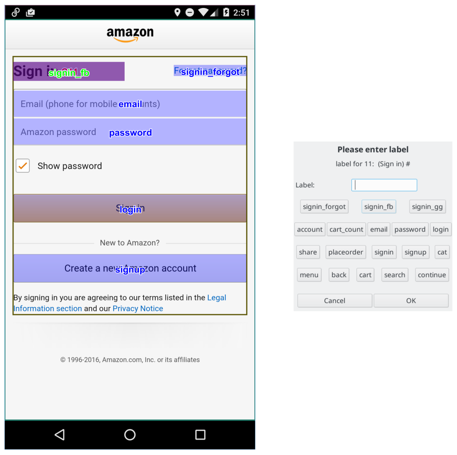


Figure 3: The GUI tool to inspect machine learning results and generate matchers. The UI of the tool is shown at left. The recognized canonical widgets have a blue rectangle overlay on them, and their labels are shown at center. A pop-up dialog to correct misclassified labels is shown at right. The possible canonical widgets are provided as buttons. To bring up the dialog, a developer clicks on a widget to select it, whose overlay becomes red, and presses the "space" key. In this example, the selected widget is incorrectly classified as "signin_fb", and this dialog asks for the correct label.

only appear on this screen, press "x" again, and enter the screen's label in an pop-up dialog. The tool then generates a matcher for this label which requires all these widgets to be present. Second, an app-specific screen can be classified as a canonical screen. Developers can fix it in a similar way, but put an app-specific screen name starting with "app_" in the dialog. The matchers generated may be further edited to check for widgets which should not exist on a canonical screen. The tool also checks the generated matchers against other screens, which prevents developers from creating a loose matcher matching unintended screens.

Alternatively, experienced developers can skip the GUI tool and directly add custom matchers to their app's configuration file:

```
@signin.login marked:'Log In'
%bookmark text:'Saved' && id:'toolbar'
```

Here a widget matcher is provided for the "login" widget on the "signin" screen. AppFlow can use it to locate this widget. Also, a screen matcher for the "bookmark" screen is provided.

Third, developers may write custom flows to test app-specific behaviors. Sometimes none of the library's flows for implementing a feature applies, so a custom flow is required for AppFlow to reach the later flows. Custom flows follow the same syntax as the flows in the test library, but they can match app-specific screens and widgets in addition to canonical ones. They can use the same properties defined in the test library or define their own ones. These custom flows will be executed alongside flows in the test library.

Lastly, developers run AppFlow to synthesize tests and generate the pass and fail results. Once developers confirm the results, AppFlow saves them for future incremental testing on each new version of the app.

If developers miss anything in the first three steps, they would see unexpected test results in the last step. Since AppFlow logs each test's execution including the flows and actions performed and the machine learning results, developers can easily figure out what is missing and repeat the above steps to fix. In our experience, we rarely need to repeat more than 10 times to test an app.

These steps are typically easy to do. The first three steps are manual and often take between half an hour to an hour in our experience applying a test library to two app categories (see §7). The most time-consuming step among them is to create custom screen and widget matchers, since developers need to navigate to different screens and carefully examine machine learning results. The steps of providing values for test variables and writing custom flows are usually straightforward. The last step takes longer (for the apps we evaluated, this step takes from one to two hours), but it is automated synthesis and requires no developer attention. After the last step has been completed once, rerunning is much faster because AppFlow saves the test results from the previous run. In all, this setup stage takes 1.5–3 hours including both manual customization and automated synthesis.

5.2 Synthesizing Full Tests

In both the first run and repeated runs, AppFlow uses the same underlying algorithm to synthesize full tests to run. It models the app behaviors as a state transition graph in which an app state is a value-assignment to all properties, including both visible properties and abstract properties. For instance, a state of a shopping app may be "$screen = detail, cart\_filled = true, loggedin = true$." The transitions of a state are the flows activated (*i.e.*, whose preconditions are satisfied by the state) at the state. Starting from the initial state, AppFlow repeatedly selects an active flow to execute, and adds the state reached by the flow to the state transition graph. It stops when it finishes exploring the entire state transition graph.

Given the state transition graph, synthesizing full tests becomes easy. To test a flow, AppFlow finds a *route* that starts from the initial state and reaches a state in which the flow is active, and combines the flows along the route and the flow to test into a full test case. As an optimization, AppFlow stores the execution time of each flow in the state transition graph, and selects the fastest route when generating full tests.

One challenge is how to reset the app to the initial state. When traversing the state transition graph, AppFlow needs to restore a previously visited state to explore another active flow in the state. AppFlow does so by uninstalling the app and cleaning up its data, and then executes the flows along the route to the state. This method fails if the app syncs its state to the server side. For instance, a flow may have added an item to the shopping cart already, and the shopping cart content is synced to the server side. When the app is re-installed, the shopping cart still contains the item. AppFlow solves this challenge by synthesizing a state cleanup route that undoes the effects of the flows to reach the state. For instance, to clean the shopping cart state, it runs the flow to remove an item from the shopping cart.

6 IMPLEMENTATION

AppFlow is implemented for the Android platform using 15,979 lines of Python code. It uses scikit-learn [68] for machine learning, and Tesseract [79] for extracting text from images.

6.1 Capturing Screen Layout

AppFlow uses the UIAutomator API [31] to capture current screen layout, a tree of all widgets with their attributes. AppFlow also captures apps' embedded webpages by communicating with apps' WebViews using the WebView Remote Debugging protocol [29]. This interface provides more details for widgets inside the embedded webpages than the UIAutomator API.

6.2 Post-Processing of the Captured Layout

The layout returned by UIAutomator contains redundant or invisible views, which would reduce the accuracy of AppFlow's screen and widget recognition. AppFlow thus post-processes the layout using several transformations, recursively applied on the layout until no more transformations can be done. For instance, one transformation flattens a container with a single child, removes empty container, and removes invisible widgets according to previously observed screens. Another transformation uses optical text recognition to find and remove hidden views. It extracts text from the area in a snapshot corresponding to each widget, and compares the text with the widget's text property. If the difference is too large, the view is marked as invisible. If all children of a widget are invisible, AppFlow marks the widget invisible, too. Our results show that this transformation safely removes up to 11.5% of the widgets.

7 EVALUATION

We focus our evaluation on the following six questions.

RQ1: How much do real-world apps share common screens, widgets, and flows and can AppFlow synthesize highly reusable flows? The amount of sharing bounds the ultimate utility of AppFlow.

RQ2: How accurately can AppFlow's machine learning model recognize canonical screens and widgets?

RQ3: How robust are the tests AppFlow synthesizes across different versions of the same app?

RQ4: How much manual labor does AppFlow save in terms of the absolute cost of creating the tests that AppFlow can readily reuse from a library?

RQ5: How much manual labor does AppFlow save in terms of the relative cost to creating a fully automated test suite for an app?

RQ6: How effectively can the tests AppFlow synthesizes find bugs? While it is out of the scope of this paper to integrate AppFlow with a production Continuous Integration system, we would like to at least apply AppFlow to the public apps on app stores and see if it finds bugs.

7.1 RQ1: Amount of Sharing Across Apps

We first manually inspected the description of all 481 apps with more than 50 million installations on Google Play [2], Android's app store, and studied whether they fall into an app category that shares common flows. Of the 481 apps, 172 are games which are known to be difficult to test automatically [20, 42], so we excluded them. In the remaining 309 apps, 196 (63.4%) of them fall into 15 categories that share many common flows, such as shopping and news. The other 113 (36.6%) apps fall into smaller categories which have larger behavior variations, such as utilities.

We conducted a deeper dive on two representative categories: shopping apps[1] and news apps. For shopping apps, we selected 40 top apps from Play Store. For news apps, we selected 20. We selected them according to the number of downloads. More than half of these apps have more than 10 million installations, and all of them have more than 1 million installations. We chose more shopping apps because they outnumber news apps in the Google Play store. Apps which cannot be automatically tested with AppFlow, such as the ones which show errors on emulators, and the ones which require SMS authentication codes, are excluded.

We created test libraries for these two categories, and found that we needed 25 canonical screens and 99 canonical widgets for the shopping apps; and 12 canonical screens and 46 canonical widgets for the news apps. These are the canonical widgets and screens required by all the flows we created based on best practices we presented in Section 4.2. We wrote 144 flows that do 111 unique feature tests (the same feature may be implemented slightly differently, requiring different flows; see §4.2) for the shopping apps; and 60 flows that does 57 unique feature tests for news apps. On average, each shopping app can reuse 61.3 (55.2%) tests and each news app can reuse 30.2 (53.0%) tests. Primarily due to an issue in UIAutomator that misses certain widgets when collecting UI layout and other implementation issues, AppFlow was able to synthesize slightly fewer tests, 61.0 tests for shopping and 28.6 for news. Figure 4 shows the histogram of the number of apps each flow can test. The average is 15.8 for shopping apps and 10.1 for news apps.

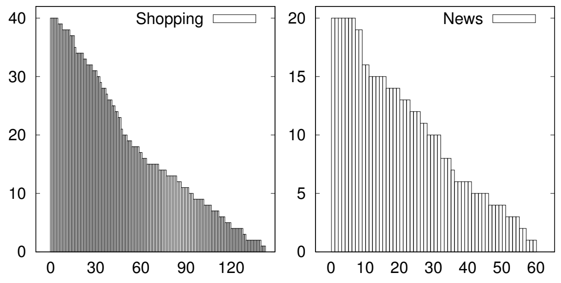


Figure 4: Number of apps each flow can test. The x-axis shows the flows, and the y-axis show the number of apps.

7.2 RQ2: Accuracy Recognizing Screens and Widgets

Our dataset consists of the tagged sample screens and widgets collected from the 40 shopping and 20 news apps. For screens with fixed content, we collected one sample per app. For screens with variable content, such as product detail screens, we collected at most five samples per app. For the 40 shopping apps, we collected 1620 screen samples which contain 9,497 canonical widgets and 55,771 non-canonical widgets. For news apps, we collected 396 screen samples which contain 1,850 canonical widgets and 9,496 non-canonical widgets.

We used well-established method leave-one-out cross-validation [6] to evaluate accuracy. Specifically, when evaluating AppFlow on one app, we trained the model on data collected from all other apps in the same category and used this app's data as the test set. This method effectively tests how our system works in real usage scenarios, where a test library is applied to test a new app which was not used during the creation of the test library.

Screen recognition accuracy. Our results show that AppFlow accurately recognized 1464 canonical screen samples for shopping apps, achieving 90.2% accuracy; and 321 canonical screen samples for news apps, achieving 81.5% accuracy. The accuracy is higher for shopping apps partly due to their larger number of samples. Averaging across all apps, the screen recognition accuracy is 87.3%.

We also evaluated the effect of feature selection in classifying screens. Using only screen layouts, the accuracy is 85.6% for the shopping apps and 75.9% for the news apps. With OCR results, the accuracy reaches 88.4% and 80.5%. With Activity name, the accuracy rises to 90.2% and 81.5%. The feature of screen layout is essential, and other features are also important.

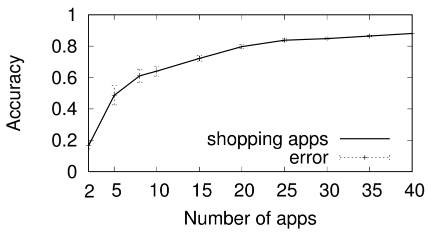


Figure 5: Accuracy vs. number of App. The x-axis shows the number of sample apps, and the y-axis shows the accuracy in screens.

Figures 5 shows how accuracy of machine learning changes with the number of sample shopping apps. We evaluated the accuracy on randomly picked subsets of sample apps. It increases with the number of apps, and reaches 80% for 20 apps. The result is similar for news apps.

Widget recognition accuracy. AppFlow's widget recognition accuracy is 88.7% for shopping apps and 85.9% for news apps, and 87.8% averaging over all evaluated apps. Similar to canonical screens, we can see that more samples result in higher widget recognition accuracy.

Figure 6 evaluates feature selection in classifying widgets. We order the features to best demonstrate their effectiveness. Using a widget's text alone can only achieve a low accuracy, while adding a widget's context and graphical features greatly improves the results. Other features, including metadata and neighbour's context, also have small contributions to the result.

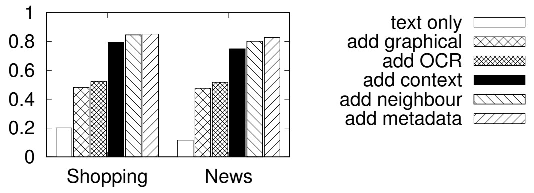
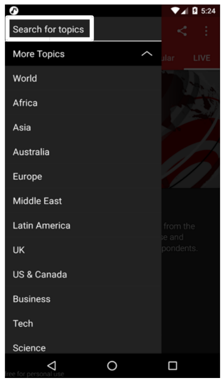


Figure 6: Features used in classifying widgets. Different bars show different combinations of features. The y-axis shows the accuracy of classifying widgets.

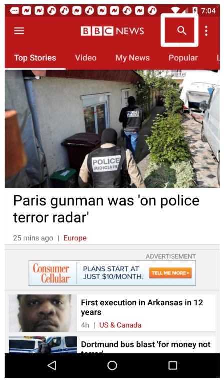
[1]Coupon and cashback apps, such as Ebates and Flipp, just serve as proxies to other businesses. Thus, they are not considered shopping apps.

7.3 RQ3: Robustness

To evaluate whether AppFlow is robust against an app's design changes, we conducted a case study with two versions of BBC News [22] whose home screens are shown in Figure 7. In the old version, there is a "Search topics" entry in its menu, as shown in the left image. Clicking on it navigates the app to the search screen, which overlaps a search input box at the top of the main screen. In the new version, the menu entry is removed. Instead, a search icon, which looks like a magnifier, appears in the toolbar of the main screen, as shown in the right image. Clicking on it still navigates the app is to the search screen, which has a new design: instead of overlapping a search input box, a separate screen is shown.



(a) Old version


BBC NEWS
Top Stories
Video
My News
Popular
Paris gunman was 'on police terror radar'
25 mins ago | Europe
ADVERTISEMENT
First execution in Arkansas in 12 years
4h | US & Canada
Dortmund bus blast 'for money not

(b) New version

Figure 7: Old and new versions of BBC News.

Using its machine learning model, AppFlow recognized canonical screens "main", "menu", and "search" and canonical widgets including the menu's search entry and the search icon. Both the flow "navigate to search screen by clicking search entry on the menu screen" and "navigate to search screen by clicking the search icon on the main screen" are common, so they are present in the test library. When we first run AppFlow on the old version and later run it on the new version, AppFlow correctly reported that the first flow turns not reusable, and the second flow becomes reusable. All the flows starting from the search screen are not affected.

7.4 RQ4: Absolute Manual Labor Savings in Creating Tests

Table 1: Customizations. Average number of lines of customization required for each app.

Number of lines	Shopping	News
Screen matchers	4.0	1.9
Widget matchers	9.1	3.8
Configuration	9.5	2.5
Custom flows	15.0	7.4
Total	37.6	15.6

AppFlow has two major costs: writing a test library for an app category and setting up testing for a specific app. Our own experience was that test library and its flows are simple and easy to write. The average number of lines of each flow is 5.7 for shopping apps, and 4.5 for news apps. Table 1 shows the number of lines of customizations required to test each specific app. On average, an app requires 30.3 lines of customizations. Only 15.9% of canonical screens and 8.9% of canonical widgets require matchers. Comparing with identifying all of them by fragile logic, AppFlow greatly increases tests' robustness.

We conducted a user study to quantify the cost saved by AppFlow. The cost of using AppFlow includes both the cost of creating a test library and applying it to a new app, so this user study targets both. The study had 15 participants. 13 of which are master students and the other two are Ph.D. students. None of them have prior knowledge of AppFlow. A state-of-the-art mobile testing framework Calabash [91] is chosen for comparison. Calabash is one of the most popular mobile testing frameworks, and its language is easy to learn and similar to AppFlow's. We randomly picked 10 test scenarios of shopping apps from 5 screens. The task is to write test scripts for these scenarios. A typical shopping app, Wish [32], is selected as sample.

Subjects are given descriptions of these scenarios and educated with the usage of AppFlow and Calabash, then asked to perform following tasks. For AppFlow, they are asked to 1) write flows 2) capture screen samples from the sample app and tag canonical widgets 3) create customizations for incorrect machine learning results of these samples 4) install the customizations and flows, run the flows, and add additional customizations if needed. These tasks are evaluating both the scenario of writing test libraries and the scenario of applying a test library to a new app. Specifically, tasks 1) and 2) are evaluating the first scenario, while tasks 3) and 4) are evaluating the second. For Calabash, they are asked to write test scripts and debug the scripts until they pass. Half of subjects follow this order, and the other half write Calabash tests first. This eliminates the effect of familiarity between systems.

We measured time spent in each task. On average, a user spends 78s in writing a flow. Tagging a screen takes 72s. Checking machine learning results and creating customizations requires 22s for one screen. Each reusable flow takes an average of 17s for developers to inspect the test result and customize the widget or screen matchers if needed. In comparison, writing and debugging a case in Calabash requires 320s.

Based on this data, we estimated the cost to create a test library. When training our model, we captured and tagged 1620 screen samples. We also wrote 144 flows for the test library. Combining with numbers above, we can calculate the test library for shopping apps takes $72s \times 1620 + 78s \times 144 \approx 35h31m$ to create.

We also estimated the cost of applying a test library to a new app. On average, 61.3 flows can be reused on an app (cf. §7.1), and 3.1 custom flows are required. The new app can have at most 25 canonical screens. Thus, applying a test library should require $22s \times 25 + 78s \times 3.1 + 17s \times 61.3 \approx 30m40s$. These numbers match our own experience. Notice that the last step in the setup stage of applying a test library is not included, because it's a mostly automatic process and the developer's time spent is insignificant compared with other steps. In contrast, using Calabash requires $320 \times 61.3 \approx 5h29m$ for creating these test cases.

These estimations show that writing test cases using AppFlow only requires 9.3% of the time when comparing with Calabash. Even if we include the time to create the test library, which should be readily available from the market, AppFlow saves cost as long as a test library is used on more than seven apps.

The cost of creating a test library depends on its complexity, the familiarity of developers with AppFlow, and the number of samples captured. Note that this is mostly one-time cost. The cost of applying a test library on a new app mainly depends on the size of it and the accuracy of machine learning models. This can further be reduced with better machine learning methods.

7.5 RQ5: Relative Manual Labor Savings to Complete Test Automation

To understand AppFlow's cost savings relative to the cost of creating a complete test automation suite, we obtained and analyzed the manual test plan of Android app JackThreads [26], a representative shopping app. This test plan is obtained directly from the developers, who were using this plan for manually testing. This typical test plan of shopping app contains 351 tests. Among them, 262 (74.6%) can be checked using test scripts. The test library of AppFlow covers 122 (46.6%) of those automatable ones. When a flow covers a test, it checks all the automatically verifiable requirements, so it is highly effective. By using AppFlow, 46.6% of the test cases can be automatically created, providing large cost savings.

There are two reasons why tests are not covered by the test library. First, test library only covers common tests, while some tests are highly specific to this app. For example, a scenario requires the images of categories on the "Categories" screen are shown as a grid, with 3 rows containing 1, 2, and 1 images. This behavior is never seen in other apps, so the test script for this scenario is not reusable by nature. Second, some scenarios refer to uncommon widgets or screens which are only present in this app. These widgets and screens are not considered canonical, thus the flows corresponding to them cannot enter the test library.

7.6 RQ6: Effectiveness in Bug Finding

Although AppFlow is evaluated on apps released on the Google Play Store, which should have been tested thoroughly, AppFlow still found multiple bugs in different apps. We found 6 bugs in shopping apps and 2 bugs in news apps. These bugs except one are not crash bugs. The non-crash bugs cannot be detected without knowing semantics, so they will be missed by tools such as DynoDroid [52] or Stoat [82]. We show 2 interesting examples.

One bug appears in the Homedepot [25] app, a shopping app for home improvements. After typing a search query into the search input box and clicking search button on soft keyboard, the app should show search results. Instead, search results appear for a second, then quickly retract. This prevents user from searching using an arbitrary keyword. On the other hand, if user click on one of search suggestions instead of the search button, it works. AppFlow detected this problem because the postcondition "screen is search results screen" failed after testing the "do a search" flow.

Another bug appears in the Groupon [24] app, a shopping app for group deals. In the search screen, if the user typed a query incorrectly and wanted to clear it, a natural way is to click the "clear search query" button, which usually looks like an "X". In this version, this does not work for the first time, but works if you click again. A human tester may miss this bug because she may think that she did not click it and tried again. AppFlow detected this bug from the failed last step in the "clear query" flow, which checks for absence of the search keyword.

8 LIMITATIONS AND FUTURE WORK

Fundamental limitations of AppFlow. AppFlow aims at greatly reducing manual effort implementing automated UI testing. We did not design AppFlow to replace manual testing completely: it is well known that as of now automated UI testing cannot replace manual UI testing completely because user experience is highly subjective [71]. However, as the advocates of Continuous Integration and DevOps articulate, early detection of bugs increases developer productivity and software quality, thereby indirectly reducing manual testing effort [57].

Along this vein, AppFlow aims at automatically testing common scenarios. Thus, the test library should only include common flows, not every possible ones. Custom flows may be written to test app-specific features. On the other hand, sufficient flows, either custom or common, must be present for AppFlow to synthesize executable tests. For instance, if there is no sign-in flow applicable, AppFlow cannot reach flows that require a user to be signed in. Our evaluation shows that only a small number of custom flows are needed in §7.4.

Flows in a test library of AppFlow should only refer to canonical widgets, which may limit checks they can perform and reduce their effectiveness. AppFlow focuses on testing core functionalities, which as we have shown are largely shared across apps and can be tested using only canonical widgets. As the test library evolves, more canonical screens can be added, and more canonical widgets can be defined, so tests can be more effective.

Machine learning misclassification. AppFlow leverages machine learning to recognize screens and widgets. Being statistical in nature, machine learning occasionally misclassifies, requiring developers to provide matchers. When an app updates, these matchers might need update as well. A flow may pass even if the feature it tests is not correctly implemented. For example, suppose a flow checks for a certain canonical widget, and a software update removes that widget, the flow may still pass if machine learning incorrectly recognized another widget as the canonical one. Machine learning misclassifications only cause problems for the simplest flows, since any flow which depends on interaction with that widget would likely break, indicating the problem to developers. However, this problem is not limited to AppFlow, because traditional test scripts typically use fragile rules to match widgets, so they have the same problem and these rules may silently fail, too. In contrast, since AppFlow uses machine learning to recognize canonical UI elements, as the accuracy of machine learning improves, this problem would also be mitigated.

Supporting other platforms. AppFlow currently only supports the Android platform. It is straightforward to use its ideas to test iOS apps. The ideas also apply to other UI testing environments, including web and desktop applications. Unlike mobile apps, Web and desktop applications tend to have more complex UIs, so recognizing UI elements might be harder. We leave these for future work.

9 RELATED WORK

Automated UI testing methods can be classified by whether they need developers' input. Random testing tools [52, 73, 87] and systematic tools [1, 3, 10, 36, 53, 54, 63, 82] explore apps' state space and detect generic problems without developers' help. Unlike AppFlow, these tools can only check for basic problems like crashes, so they cannot test if apps can complete scenarios.

Other methods need developers to specify expected behaviors. Model based testing [35, 58, 84, 88] requires models or UI patterns, which have to be created manually for each app. These models are usually hard to write and maintain. PBGT [12] aims to reduce the effort of modeling by reusing models, but a model created using it is highly specific to an app and usually not reusable on other apps.

Concurrent to our work, Augusto [47] generates semantic UI tests based on popular functionalities. It explores an application with GUI ripping, matches the traversed windows with UI patterns, verifies them according to semantic models defined using Alloy [41], and generates semantic tests. We share the same intuition that apps implement common tests (called application independent functionalities, or AIFs, in Augusto) using common UI elements, and we both generate semantic tests. There are also key differences. At the technical level, unlike Augusto which uses rules to match widgets and screens, AppFlow uses machine learning methods to recognize them, which are more robust. AppFlow discovers reusable flows by evaluating flows on an app and progressively constructing a state transition graph, while Augusto dynamically extracts an application's GUI model, identifies AIFs inside it, and generates tests for them. At the experimental level, we conducted studies of real-world apps to quantify the amount of sharing across apps in the same category. In addition, two posters [5, 72] discussed the potential of transferring tests written for an app to another.

Script-based testing frameworks, such as Calabash [91], Espresso [30], and others [17, 27, 44] require developers to write and maintain test scripts. As we mentioned in section 1, these scripts require considerable efforts to write and maintain. This prevents companies from adopting such methods. Specifically, these scripts use fragile rules to find UI elements, which makes them not robust to UI changes and increases maintenance cost.

Test record and replay [21, 23, 27, 33, 37, 45, 70, 75] eases test writing. Like other scripts, tests generated by it usually refer to UI elements with absolute position [21, 28, 33, 37, 70] or fragile rules [23, 75]. These scripts produce unstable results and may not adapt to different screen resolutions [45]. Worse, these rules may match widgets with properties not intended by developers, further reducing robustness. AppFlow enables scripts to be robust and reused by using machine learning to locate UI elements and using its synthesis system to automatically discover an app's behavior. This greatly reduces the cost of adopting automatic testing.

Sikuli [8] uses computer vision to help developers and enables them to create visual test scripts. It allows developers to use images to define widgets and expected feedbacks, and then matches these images with screen regions to find widgets and check assertions. It can also record visual tests and replay them. Similar to Sikuli, AppFlow also uses computer vision in recognizing UI elements, but AppFlow also combined non-visual features from UI elements which are essential for correct recognition. AppFlow's model is trained on samples from multiple apps, which enables AppFlow to adapt to UI changes and recognize same UI element in different apps. Unlike Sikuli which may only adapt to spatial changes in UI elements, AppFlow can adapt to behavior changes which may result in addition and removal of UI elements.

UI test repair [11, 34, 38, 61] aims at reducing test maintenance cost, by automatically fixing UI tests after applications' designs change. They find alternative UI event sequences for UI tests under repair to keep them runnable. Although these method are efficient, they can only fix a portion of all the broken tests, while the remaining ones still need manual work.

Some previous works create models or tests automatically. UnitPlus [80] and other works [46, 62, 78, 92, 93] used available tests to assist developers in creating new tests for the same app, but tests still need to be created first. GK-Tail+ [56] and other work [16] create models or tests by mining traces. Polariz [55] uses a crowd with no testing experience to provide test cases and mines common patterns among multiple apps. AppFlow can be combined with these works to free developers from writing test libraries. Previous works [59, 60] generate test cases from well-defined operations with automatic planning, while AppFlow generates tests by progressively discover an app's behavior, which is necessary to handle different app designs and synthesize only tests reusable in this app.

Machine learning algorithms has been widely used in software engineering. Previous works [15, 48, 51, 65, 74, 77, 81, 83, 85, 89, 90] learn useful features from codes for code completion, clone detection, bug finding, similar app detection, etc. Poster [72] uses off-the-shelf model only to calculate text similarity between UI elements. To the best of our knowledge, AppFlow is the first work to apply machine learning in recognizing apps' screens and widgets.

10 CONCLUSION

In this paper we presented AppFlow, a system for synthesizing highly robust, highly reusable UI tests. AppFlow achieves this by realizing that apps in the same category share much commonality. It leverages machine learning to recognize canonical screens and widgets for robustness and reusability, and provides a system for synthesizing complete tests from modular tests of main functionalities of an app category.

We evaluated AppFlow on 60 popular apps in the shopping and the news category, two case studies on the BBC news app and the JackThreads shopping app, and a user-study of 15 subjects on the Wish shopping app. Results show that AppFlow accurately recognizes screens and widgets, synthesizes highly robust and reusable tests, covers 46.6% of all automatable tests for Jackthreads with the tests it synthesizes, and reduces the effort to test a new app by up to 90%. It also found eight bugs in the evaluated apps, which were publicly released and should have been thoroughly tested. Seven of them are functionality bugs.

ACKNOWLEDGEMENTS

We thank Gail Kaiser, Jonathan Bell, and the anonymous reviewers for their valuable comments. This work was funded in part by NSF CNS-1564055, ONR N00014-16-1-2263, and ONR N00014-17-1-2788 grants.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32, 5 (2015), 53–59.

[2] androidrank.org. 2018. Android application ranklist - All applications. (2018). https://www.androidrank.org/listcategory?category=&sort=4&price=all

[3] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 641–660.

[4] Nick Babich. 2018. 10 Do's and Don'ts of Mobile UX Design. http://theblog.adobe.com/10-dos-donts-mobile-ux-design/. (Feb. 2018).

[5] Farnaz Behrang and Alessandro Orso. 2018. Automated Test Migration for Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. 384–385.

[6] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[7] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*. ACM, 144–152.

[8] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. 1535–1544.

[9] David Chelimsky, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. 2010. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends.*

[10] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA '13)*. 623–640.

[11] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. WATER: Web Application TEst Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering (ETSE '11)*. 24–29.

[12] Pedro Costa, Ana CR Paiva, and Miguel Nabuco. 2014. Pattern based GUI testing for mobile applications. In *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*. IEEE, 66–74.

[13] Wei Dai and Jeffrey Walton. 2018. Crypto++ Library 5.6.5 | Free C++ Class Library of Cryptographic Schemes. https://www.cryptopp.com/. (2018).

[14] Navneet Dalal and Bill Triggs. 2005. Histograms of oriented gradients for human detection. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005.*, Vol. 1. IEEE, 886–893.

[15] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. 2012. XIAO: Tuning Code Clones at Hands of Engineers in Practice. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. 369–378.

[16] Markus Ermuth and Michael Pradel. 2016. Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. 82–93.

[17] JS Foundation. 2018. Appium: Mobile App Automation Made Awesome. http://appium.io/. (2018).

[18] OpenSSL Software Foundation. 2017. OpenSSL. https://www.openssl.org/. (2017).

[19] The Apache Software Foundation. 2017. ZooKeeper. https://zookeeper.apache.org/. (2017).

[20] Jerry Gao, Xiaoying Bai, Wei-Tek Tsai, and Tadahiro Uehara. 2014. Mobile Application Testing: A Tutorial. *Computer* 47, 2 (Feb. 2014), 46–55.

[21] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing- and Touch-sensitive Record and Replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 72–81.

[22] Google. 2018. BBC News. https://play.google.com/store/apps/details?id=bbc.mobile.news.ww. (2018).

[23] Google. 2018. Espresso Test Recorder. https://developer.android.com/studio/test/espresso-test-recorder.html. (June 2018).

[24] Google. 2018. Groupon - Shop Deals & Coupons. https://play.google.com/store/apps/details?id=com.groupon. (2018).

[25] Google. 2018. The Home Depot. https://play.google.com/store/apps/details?id=com.thehomedepot. (2018).

[26] Google. 2018. JackThreads: Men's Shopping. https://play.google.com/store/apps/details?id=com.jackthreads.android. (2018).

[27] Google. 2018. monkeyrunner. (June 2018). http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[28] Google. 2018. monkeyrunner. http://developer.android.com/tools/help/MonkeyRunner.html. (June 2018).

[29] Google. 2018. Remote Debugging Webviews | Web | Google Developers. https://developers.google.com/web/tools/chrome-devtools/remote-debugging/webviews. (July 2018).

[30] Google. 2018. Testing UI for a Single App. https://developer.android.com/training/testing/ui-testing/espresso-testing.html. (May 2018).

[31] Google. 2018. Testing UI for Multiple Apps | Android Developers. https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html. (May 2018).

[32] Google. 2018. Wish - Shopping Made Fun. https://play.google.com/store/apps/details?id=com.contextlogic.wish&hl=en. (2018).

[33] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (ISPASS '15)*. 215–224.

[34] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: An Incremental Approach for Repairing Record-replay Tests of Web Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 751–762.

[35] Benedikt Hauptmann and Maximilian Junker. 2011. Utilizing user interface models for automated instantiation and execution of system tests. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*. ACM, 8–15.

[36] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*.

[37] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile Yet Lightweight Record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 349–366.

[38] Si Huang, Myra B. Cohen, and Atif M. Memon. 2010. Repairing GUI Test Suites Using a Genetic Algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. 245–254.

[39] Amazon Web Services Inc. 2018. What is DevOps? - Amazon Web Services (AWS). https://aws.amazon.com/devops/what-is-devops/. (2018).

[40] ThoughtWorks Inc. 2018. Continuous integration | ThoughtWorks. https://www.thoughtworks.com/continuous-integration. (2018).

[41] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290.

[42] Jouko Kaasila. 2015. Mobile Game Test Automation Using Real Devices. https://developers.google.com/google-test-automation-conference/2015/presentations#Day1LightningTalk2. (Dec. 2015).

[43] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference for Learning Representations (ICLR 2015)*.

[44] Edmund Lam, Peilun Zhang, and Bor-Yuh Evan Chang. [n. d.]. ChimpCheck: Property-based Randomized Test Generation for Interactive Apps. *(Onward! 2017)*.

[45] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and Replay for Android: Are We There Yet in Industrial Cases?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 854–859.

[46] Mathias Landhäusser and Walter F. Tichy. 2012. Automated Test-case Generation by Cloning. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST '12)*. 83–88.

[47] Daniele Zuddas Leonardo Mariani, Mauro Pezzè. 2018. Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*.

[48] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. 306–315.

[49] Cucumber Limited. 2018. Cucumber. https://cucumber.io/. (2018).

[50] Cucumber Limited. 2018. Gherkin Syntax: Cucumber. https://docs.cucumber.io/gherkin/. (2018).

[51] M. Linares-Vásquez, A. Holtzhauer, and D. Poshyvanyk. 2016. On automatically detecting similar Android apps. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10.

[52] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. 224–234.

[53] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.

[54] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. 94–105.

[55] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd Intelligence Enhances Automated Mobile Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. 16–26.

[56] Leonardo Mariani, Mauro Pezze, and Mauro Santoro. 2017. GK-Tail+: An Efficient Approach to Learn Precise Software Models. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*.

[57] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. 233–242.

[58] Atif M Memon. 2007. An event-flow model of GUI-based applications for testing. *Software Testing Verification and Reliability* 17, 3 (2007), 137–158.

[59] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. 1999. Using a Goal-driven Approach to Generate Test Cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. 257–266.

[60] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. 2001. Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Trans. Softw. Eng.* 27, 2 (2001), 144–155.

[61] Atif M. Memon and Mary Lou Soffa. 2003. Regression Testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*. 118–127.

[62] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Leveraging Existing Tests in Automated Test Generation for Web Applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. 67–78.

[63] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 559–570.

[64] Hacker News. 2015. https://news.ycombinator.com/item?id=9293445. (March 2015).

[65] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API Code Recommendation Using Statistical Learning from Fine-grained Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 511–522.

[66] Diego Ongaro and et al. 2018. Raft Consensus Algorithm. https://raft.github.io/#implementations. (July 2018).

[67] Diego Ongaro and John K Ousterhout. 2014. In Search of an Understandable Consensus Algorithm.. In *USENIX Annual Technical Conference*. 305–319.

[68] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[69] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 33.

[70] Z. Qin, Y. Tang, E. Novak, and Q. Li. 2016. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) (ICSE '16)*. 571–582.

[71] Rudolf Ramler and Klaus Wolfmaier. 2006. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. In *Proceedings of the 2006 International Workshop on Automation of Software Test (AST '06)*. 85–91.

[72] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Efficient GUI Test Generation by Learning from Tests of Other Apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. 370–371.

[73] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. 2014. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 190–203.

[74] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 419–428.

[75] Renas. 2016. Robotium framework for test automation. http://www.robotium.org. (Sept. 2016).

[76] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988. Learning representations by back-propagating errors. In *Neurocomputing: Foundations of Research*. Chapter Learning Representations by Back-propagating Errors, 696–699.

[77] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 1157–1168.

[78] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (Sept 2016), 805–824.

[79] R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02 (ICDAR '07)*. 629–633.

[80] Yoonki Song, Suresh Thummalapenta, and Tao Xie. 2007. UnitPlus: Assisting Developer Testing in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '07)*. 26–30.

[81] Fang-Hsiang Su, J. Bell, G. Kaiser, and S. Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10.

[82] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 245–256.

[83] Jeffrey Svajlenko and Chanchal K. Roy. 2017. CloneWorks: A Fast and Flexible Large-scale Near-miss Clone Detection Tool. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. 177–179.

[84] Tommi Takala, Mika Katara, and Julian Harty. 2011. Experiences of system-level model-based GUI testing of an Android application. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 377–386.

[85] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 11–20.

[86] OpenCV team. 2018. OpenCV library. http://opencv.org/. (2018).

[87] UI/Application Exerciser Monkey 2018. UI/Application Exerciser Monkey. (June 2018). http://developer.android.com/tools/help/monkey.html.

[88] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. 2006. Automation of GUI testing using a model-driven approach. In *Proceedings of the 2006 international workshop on Automation of software test*. ACM, 9–14.

[89] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 87–98.

[90] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. 1981. The Effect of Modularization and Comments on Program Comprehension. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. 215–223.

[91] Xamarin. 2018. Calaba.sh - Automate Acceptance Testing for iOS and Android Apps. http://calaba.sh/. (2018).

[92] R. Yandrapally, G. Sridhara, and S. Sinha. 2015. Automated Modularization of GUI Test Cases. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 44–54.

[93] Tianyi Zhang and Miryung Kim. 2017. Automated Transplantation and Differential Testing for Clones. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 665–676.