Automatically Translating Bug Reports into Test Cases for Mobile Apps

Mattia Fazzini
Georgia Tech
Atlanta, GA, USA
mfazzini@cc.gatech.edu

Martin Prammer
Georgia Tech
Atlanta, GA, USA
mprammer3@gatech.edu

Marcelo d'Amorim
Federal U. of Pernambuco
Recife, Brazil
damorim@cin.ufpe.br

Alessandro Orso
Georgia Tech
Atlanta, GA, USA
orso@cc.gatech.edu

ABSTRACT

When users experience a software failure, they have the option of submitting a bug report and provide information about the failure and how it happened. If the bug report contains enough information, developers can then try to recreate the issue and investigate it, so as to eliminate its causes. Unfortunately, the number of bug reports filed by users is typically large, and the tasks of analyzing bug reports and reproducing the issues described therein can be extremely time consuming. To help make this process more efficient, in this paper we propose Yakusu, a technique that uses a combination of program analysis and natural language processing techniques to generate executable test cases from bug reports. We implemented Yakusu for Android apps and performed an empirical evaluation on a set of over 60 real bug reports for different real-world apps. Overall, our technique was successful in 59.7% of the cases; that is, for a majority of the bug reports, developers would not have to study the report to reproduce the issue described and could simply use the test cases automatically generated by Yakusu. Furthermore, in many of the remaining cases, Yakusu was unsuccessful due to limitations that can be addressed in future work.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Mobile testing and debugging, natural language processing

ACM Reference Format:
Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically Translating Bug Reports into Test Cases for Mobile Apps. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3213846.3213869

1 INTRODUCTION

Due to the inherent limitations of software verification techniques, it is virtually impossible to eliminate all faults from a software system before releasing it. It is therefore unavoidable that users will experience software failures. For this reason, most software systems provide a way for users to submit bug reports, either automatically (e.g., in the case of crashing bugs) or manually. The main goal of a bug report is to collect information about the problems experienced by the users, so that developers can investigate these issues, find their causes, and eliminate such causes [20, 29]. In order to do so, developers normally have to look at a bug report, understand what steps led to the issue reported, try to reproduce these steps and the corresponding issue, and debug the issue. Unfortunately, performing these tasks can be extremely time consuming, especially in the presence of a large number of bug reports and of reports with incomplete information [4].

To help make this process more efficient, we propose Yakusu,[1] a technique for generating executable UI test cases from bug reports through a combination of program analysis and natural language processing techniques. Specifically, Yakusu takes as input an app and a bug report and operates in three main phases. First, it analyzes the app to identify the elements available in the UI and generate an ontology for the app. Second, it analyzes the bug report, leveraging the generated ontology, and tries to identify a set of steps provided by the user for reproducing the reported issue. Finally, if successful, it tries to generate a test case that reproduces the issue by mapping the identified steps to actual UI events.

In defining Yakusu, we had to overcome two main challenges. First, extracting structured information from typically non-structured data (i.e., bug reports) is a non-trivial task, as it involves interpreting possibly incomplete descriptions that use a broad, imprecise, and context-dependent language (e.g., "Start a new post"—see Section 2). Second, even when the steps provided by the user in the bug report have been correctly identified, there is typically a logical gap between such steps and actual UI events, and the sequence of steps may be incomplete. Therefore, generating a test case that suitably encodes the bug report often involves searching a large space of possible solutions (i.e., sequences of events).

Although the problem of synthesizing code from natural language descriptions has been studied before (e.g., [5, 27, 47]), most existing techniques in this area either make strong assumptions on the format of the textual description or do not consider the fact that steps might be described at different levels of abstraction. The format issue is particularly relevant in the domain of bug reports, where end users typically do not have the experience or technical expertise to structure the list of steps in a way that makes them easily consumable by developers (or by a tool). Furthermore, techniques based on learning (e.g., [5, 27]) tend to perform poorly in the presence of sentences and terms that are specific to an app

[1] Yakusu means "to translate" in Japanese.

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213869

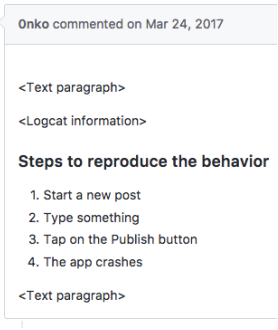

Figure 1: Bug report.

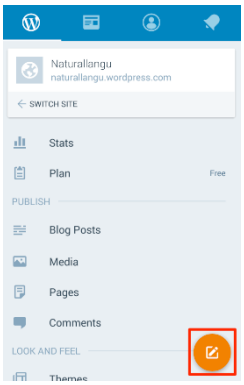

Figure 2: Main screen.

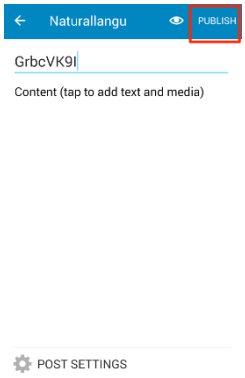

Figure 3: Edit post screen.

```
1 public void testReproduce()
2 {
3   // click action
4   onView(
5     withId(
6       R.id.fab_button))
7     .perform(
8       click());
9   // type action
10  onView(
11    withId(
12      R.id.post_title))
13    .perform(
14      typeText("GrbcVK9I"));
15  // click action
16  onView(
17    withId(
18      R.id.menu_save_post))
19    .perform(
20      click());
21 }
```

Figure 4: Test encoding.

and rarely appear in other reports. YAKUSU, conversely, can handle these cases by leveraging the ontology of the app that it generates and using it to match these app-specific terms. The use of an ontology also helps YAKUSU handle reports without assuming a specific format, as the ontology allows the technique to match words in a sentence to elements in the UI and then infer the corresponding actions by analyzing the elements' behavior at runtime.

To assess the effectiveness of our technique, we implemented YAKUSU for Android apps and evaluated it on over 60 real bug reports for a range of real-world Android apps. Overall, our technique was able to generate a test case that reproduced the issue described in the bug report in 59.7% of the cases, which provides initial, yet clear evidence of the usefulness of YAKUSU. If confirmed by additional studies, these results would indicate that, in almost 60% of the cases, developers could simply use the test cases automatically generated by YAKUSU and would have to look at the corresponding bug reports only if they need additional details. Further, the limitations that prevented YAKUSU from generating tests for many of the remaining reports could be addressed with a combination of further research and careful engineering. It is also worth noting that, although we focused on Android applications in this paper, the general approach could be applied to other mobile platforms.

This paper makes the following contributions:

- An automated technique for generating, from bug reports written in natural language, test cases that reproduce the issues described in such reports.
- An implementation of the technique for Android apps that is publicly available, together with the artifacts and infrastructure we used in our evaluation [13].
- An empirical evaluation that provides initial evidence of the effectiveness of our approach.

2 TERMINOLOGY & MOTIVATING EXAMPLE

This section introduces some relevant terminology and presents an example that we use to motivate and illustrate our technique.

2.1 Terminology

Given a bug report B that describes a failure (or issue) F for an app A, we informally use the terms *relevant failure* and *relevant app* to indicate F and A. We also use the terms *abstract step* or *abstract action* interchangeably to indicate the description, in B, of an operation that has to be performed to reproduce F (e.g., create a user). Finally, we use the term *UI action* to indicate an actual action that can be performed on the UI of A (e.g., push button "New User").

2.2 Motivating Example

Our motivating example is a bug report for WORDPRESS [51], a widely used real-world app for creating web sites and blogs that has been installed over 5 million times. Figure 1 shows the bug report as it appears in WORDPRESS's issue tracking system [1]. The report contains, under the header "Steps to reproduce the behavior", a list of three abstract actions followed by a description of the failure. Figures 2 and 3 show the screens traversed when performing the actions listed in the report.

As this example shows, actions can be described at different levels of abstraction; some actions refer directly to easily identifiable UI elements, whereas for other actions there is a logical gap between their description and the corresponding actual UI actions. Abstract action"Tap on the Publish button", for instance, can be easily mapped to the UI action of clicking the button labeled "PUBLISH" in the screen depicted in Figure 3 (top right, highlighted). Conversely, the abstract action "Start a new post" corresponds to clicking the round button in the screen shown in Figure 2 (bottom right, highlighted). In this case, identifying the corresponding UI action requires a deeper analysis of both the report and the app.

As we describe in detail in Section 3, YAKUSU uses a combination of program analysis and natural language processing techniques to identify mappings from abstract actions to UI actions and generate a test case that reproduces the relevant failure. For this example, the test that would be generated by YAKUSU is shown in Figure 4. The test is encoded using the Espresso framework [18], which uses app identifiers to refer to UI elements in the app (see Section 4 for details). Lines 4–8 encode the action "Start a new post" and correspond to clicking the aforementioned round button in Figure 2 (element with identifier `fab_button` in the app). Then, lines 10–14 encode the action "Type something" and consist of typing some randomly-generated text on the text box with label "Title" in Figure 3 (element with identifier `post_title` in the app). Finally, lines 16–20 encode the action "Tap on the Publish button" by clicking, as described above, the button in the top-right corner of Figure 3 (element with identifier `menu_save_post` in the app).

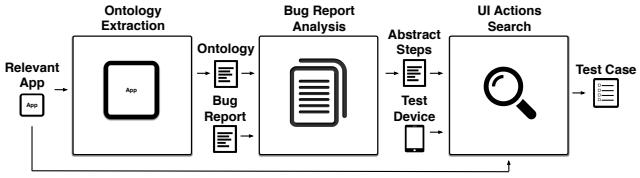

Figure 5: High-level overview of the technique.

3 TECHNIQUE

This section presents YAKUSU, a technique for translating bug reports written in natural language into test cases. First, YAKUSU combines static analysis and natural language processing to extract, from a given bug report, a list of abstract steps describing how to reproduce the relevant failure. Then, YAKUSU performs a dynamic search, guided by the previously extracted list of abstract steps, to find a set of UI actions that match these steps.

Figure 5 provides an overview of YAKUSU's workflow, which consists of three phases. The *ontology extraction* phase takes the relevant app as input and produces as output the app ontology, which describes the elements available in the UI. The *bug report analysis* phase takes as input the ontology and the bug report and produces as output a list of abstract steps that the execution needs to follow to reproduce the failure documented in the report. Conceptually, the ontology allows for binding the vocabulary used in the bug report with the elements in the UI. This binding is important for producing the list of abstract steps. Finally, the *UI actions search* phase takes as input the list of abstract steps and produces a concrete test case that reproduces the relevant failure. In this phase, the technique runs the relevant app on a test device and tries to map the list of abstract steps provided as input into UI actions. When all abstract steps are mapped, YAKUSU encodes the identified UI actions as a test case that can be run on the app, which is the final output of the technique. The rest of this section describes each phase in detail.

3.1 Ontology Extraction

This phase extracts a machine-comprehensible description of the UI elements; this description supports the construction of the mapping between the vocabulary used in the bug report and the UI elements in the relevant app, which takes place in the next phase (see Section 3.2). To create this description, YAKUSU statically analyzes the app using a two-pronged approach: first, it analyzes the UI configuration files to identify the widgets that are created statically; then, it analyzes the source code of the app to identify additional widgets that are created at runtime. For this latter analysis, YAKUSU builds a graph of the app that models the creation of UI elements and the modification of their state through setters. Nodes in the graph represent creation and modification operations. Edges represent def-use relations between such operations. Using this graph, YAKUSU can therefore identify (1) which UI elements are dynamically created and (2) what their properties are. For instance, YAKUSU would be able to identify the creation of a button, its label, the callbacks associated with the button, and so on.

YAKUSU stores the identified UI elements, together with (some of) their properties, as tuples. The set of these tuples constitutes the ontology for the app. Specifically, YAKUSU stores in the ontology three kinds of properties for a given UI element: (1) its *label*, (2) the name of the file that contains its associated *icon*, and (3) its *identifier*. (If one or more of these properties is not present, YAKUSU simply stores an empty value for it.)

We selected these three properties because they are particularly suitable for characterizing and identifying a UI element, as we now illustrate. For an element that can display a label, users tend to use such label to refer to this element in bug reports. As an example, consider the step "Tap on the Publish button" in the example of Section 2, which uses the label "Publish" of the button to refer to it. Similarly, for a UI element represented by an icon, users often use the name of the object represented by the icon to refer to that element. In this case, there is no textual property to store, so YAKUSU stores in the tuple the name of the file that contains the icon, under the assumption that such name is representative of the icon. As an example, consider the step "Press on attach", from one of the bug reports we used in our evaluation (Section 5). In this case, "attach" refers to a paper clip icon whose corresponding filename is `attachFileImage`. Finally, users may refer to an element of the UI based on its functionality, which may not be reflected in the visual aspect of the element. Using the identifier of an element in the app allows YAKUSU to handle some of these cases, as developers often define identifiers based on the functionality of their corresponding element. The step "Select a Client", for instance, is present in another bug report from our evaluation and is used to refer to one of the elements in a list of clients. Because the identifier for a client is `tv_clientName`, YAKUSU is able to match that step with the correct element using its identifier.

It is worth noting that labels are stored unchanged, whereas YAKUSU performs some normalization for properties icon and identifier to facilitate the analysis in the following phases of the technique. In particular, our technique replaces underscores with spaces and splits apart composite words that follow a camel case convention, both of which are common occurrences in apps [40].

3.2 Bug Report Analysis

This phase aims to extract from a bug report the sequence of abstract steps to be performed on the UI of the relevant app for reproducing the relevant failure described in the report. Because bug reports are typically written in natural language, YAKUSU analyzes their content using natural language processing (NLP) techniques, translating the text into dependency trees [11, 25]. (A dependency tree is a directed graph that captures the syntactic structure of a sentence and provides a representation of grammatical relations between words in the sentence.) The tree is characterized by a root word and by relations that connect pair of words in the sentence. Two words involved in a relation are also defined as head and dependent, with the direction of the relation going from the head to the dependent. Our technique uses dependency trees based on the Universal Dependency schema [48]. In this schema, frequently used relations can be broken into two sets: *clausal relations*, which describe syntactic roles with respect to a predicate (often a verb), and *modifier relations*, which categorize how dependents can modify their heads. Figure 6 provides an example of a dependency tree. In the remainder of this section, we provide more details on how the dependency tree is computed and processed with the help of Algorithm 1.

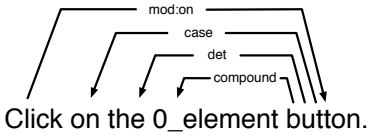

Figure 6: Dependency tree computed by Yakusu.

The algorithm takes as input the ontology of the relevant app (*ontology*) and the text of the bug report (*br*), and produces as output a list of abstract steps (*aSteps*) that represent the list of steps described in the report. An abstract step is a tuple ⟨*action*, *target*, *props*⟩, where *action* is a word describing the UI action to be performed, *target* is a list of words describing the UI element to be exercised by *action*, and *props* is the list of properties affecting the behavior of *action*. The algorithm begins with an empty list of abstract steps (line 2) and starts by analyzing the bug report (function Get-A-Steps-Text) to identify the portion of text that describes how to reproduce the relevant failure. Get-A-Steps-Text uses a set of heuristics defined based on guidelines on how to report bugs for mobile apps [4, 15, 20, 29]. More precisely, our technique first checks whether the bug report has a section whose content or heading contains the lemma "reproduce". If so, it considers the identified section as the one describing how to reproduce the relevant failure (as it is common practice to use this lemma to describe such section). If such a section is not present, Yakusu tries to identify the text describing abstract steps by checking whether the report contains a list of bullet points. If neither the section nor the bullet points are present, our technique considers the complete text of the report as relevant. For the bug report of Figure 1, our technique would identify the section with header "Steps to reproduce the behavior", which contains lemma "reproduce", as the text describing the sequence of abstract steps needed to reproduce the failure.

After identifying the right portion of text, Yakusu preprocesses such text (function Preprocess-Text) to simplify the subsequent analysis. Preprocess-Text performs three standard NLP operations: noise removal, lexicon normalization, and object standardization [34]. Our technique, however, specializes these operations to the domain of bug reports for mobile apps. First, for *noise removal*, Yakusu discards content within parenthesis, which is in our experience unnecessary for reproduction. For instance, the sentence "*(don't add anything but just)* Press menu button and Reconnect" (from one of the benchmarks in Section 5) is changed to "Press menu button and Reconnect". Second, for *lexicon normalization*, Yakusu normalizes non-standard words to their canonical form to simplify parsing and understanding of actions. (We identified such words from a set of more than 400 tutorials [13] on mobile apps that we collected on the web, by manually inspecting the tutorials for words that are not app specific and refer to actions on the UI.) As an example, in the bug report of Figure 1, Yakusu changes "*Tap* on the Publish button" to "*Click* on the Publish button". Finally, for *object standardization*, the technique simplifies the text that refers to the target of an action by leveraging the fact that the text displayed by an app usually follows a title or sentence-case convention [44]. Specifically, Yakusu tries to identify sequences of words in title or sentence-case format that are also textual properties in the ontology of the app and (1) replaces such sequences with a freshly created textual reference ID and (2) adds to *refMap* the mapping between

Algorithm 1: Bug report analysis.

```
   Input  : ontology: ontology of the app
            br: text of the bug report
   Output : aSteps: list of abstract steps describing how to reproduce the relevant failure
1  begin
2      aSteps = []
3      refMap = ∅
4      text = Get-A-Steps-Text(br)
5      text = Preprocess-Text(text, ontology, refMap)
6      foreach sentence ∈ text do
7          foreach clause ∈ sentence do
8              tree = Get-Dependency-Tree(clause)
9              root = Get-Root(tree)
10             if root ∈ {click, type, scroll, swipe, rotate, ...} then
11                 action = root
12                 target = Extract-Target(action, tree, refMap)
13                 props = Extract-Props(action, target, tree, refMap)
14                 aStep = Create-A-Step(action, target, props)
15                 aSteps.Add(aStep)
16             else
17                 if Semantically-Related(clause, ontology, refMap) then
18                     gAStep = Create-Generic-A-Step(null, clause, [])
19                     aSteps.Add(gAStep)
20     return aSteps
```

the ID and the original text in the ontology. For the bug report from Section 2, Yakusu transforms the sentence "Click on the *Publish* button" into "Click on the *0_element* button" and associates "0_element" to text "Publish" in *refMap*. This preprocessing step is particularly useful when the text being replaced contains multiple words, as it simplifies the later analysis of dependency trees.

After preprocessing the text of the bug report, Yakusu enters its main loop (lines 6–19), where it analyzes each sentence in the text. More specifically, it analyzes each clause that appears in a sentence, as each of them can specify a different action on the UI. For example, the sentence "Press menu button and Reconnect" contains two clauses connected by a coordinating conjunction, and each clause specifies a different action: (1) open the menu of the app and (2) perform the "Reconnect" action. To identify clauses from sentences, our technique leverages related work [2] that parses a dependency parse tree recursively and, at each step, predicts whether an edge should yield an independent clause.

For each clause, our algorithm computes the clause's dependency tree (Get-Dependency-Tree). Figure 6 provides, as an example, the dependency tree for clause "Click on the 0_element button". The algorithm analyzes the root word of the tree (e.g., "Click" in Figure 6) to assess whether it refers to a UI action using, as we discussed earlier, the domain knowledge we distilled from 400 tutorials (Line 10 in Algorithm 1). If the root word of the dependency tree refers to a UI action, Yakusu further analyzes the tree to extract the properties of the action and the action target, which is the element in the UI affected by the action and, for certain actions, can be missing. Similarity, the list of properties of an action can be empty. For example, the clause "Type something" from the example of Section 2 does not specify the target of the action. In this case, our technique would consider any editable UI element as a possible target for the action.

When analyzing a dependency tree, Yakusu looks for two types of information: the list of words representing the target (Extract-Target) and the list of properties affecting the behavior of the action (Extract-Props). To identify words that are affected by the action (i.e., the target of the action), our technique analyzes the subtrees rooted at (1) core dependent relations and (2) non-core dependent relations that are associated with the root word

(the action) of the dependency tree. In the case of the dependency tree of Figure 6, the technique would identify as the target of the action the subtree rooted at the word associated with the nominal modifier relation (`nmod:on`). The subtree includes the words "on the 0_element button". Yakusu further analyzes the subtree and, if it contains a textual reference (identified using *refMap*), it uses the text associated with the reference as the target of the action ("Publish", in the case of the example). Otherwise, it uses as target all the words in the subtree. To identify words that detail the behavior of an action, the technique analyzes modifier relations, core dependent relations, and non-core dependent relations [48] associated with the root word (the action) of the dependency tree. For example, in the clause "long click retweet icon underneath tweet" (from one of the benchmarks in Section 5), the action "click" is affected by the word "long" through an adverbial modifier relation. Yakusu identifies such relation, captures the precise action (`long click`) to be performed on the UI (line 11), and stores this information in the list of properties of the action. After the technique determines the action (*action*), the target (*target*), and the properties (*props*) from the clause, it encodes this information into an abstract step (Create-A-Step) and stores the step in the abstract steps list (line 15). We call the encoding of action, target, and properties an *abstract step* because the following phase of the technique will try to find how to concretely execute the step on the relevant app (i.e., find the corresponding concrete UI action).

When there is a logical gap between the description contained in the clause and the corresponding actions to be performed on the UI (i.e., the root word is not *click*, *type*, *scroll*, and so on), the technique assesses whether the clause relates to an element of the UI by semantically comparing (Semantically-Related) the content of the clause with the elements in the ontology of the relevant app. If the clause is semantically related to an element of the UI, Yakusu treats the clause as the target of an abstract step as in some cases users describe actions in terms of the elements of the UI. The clause "Start a new post" (from the motivating example in Section 2) is an example of such situation. More precisely, Yakusu compares the content of the clause with the components (text, icon, identifier) of the tuples in the ontology using word embeddings computed from a word2vec model [32, 33], which offers a mathematical representation of the meaning of a word. Specifically, word2vec produces a vector space from a large corpus of text, where each word in the corpus is assigned to a vector in the space. Vectors are positioned in the space such that words that share common contexts are located in close proximity to one another. The technique computes the word2vec model from a corpus of 100 billion words [19], as word vectorization requires training on very large sets of words [35]. Yakusu represents the clause as a vector computed by averaging the vectors of the words in the clause, after removing stopwords as they introduce unnecessary noise. By taking the average, the technique is able to incorporate the meaning of every word in the vector representation of the clause. Yakusu computes the vector of the components in the tuples of the ontology in the same way. It then compares the vector of the clause with the vector of each component by computing the cosine similarity of the two vectors. The similarity value ranges between $[-1.0, 1.0]$, where 1.0 corresponds to the highest similarity. The technique considers a clause to be semantically related to the a UI element if their cosine similarity value is greater than 0.5. We computed this value empirically using a set of bug reports considered for training purposes; this set of bug reports was not used in the evaluation of Section 5.

For example, Yakusu associates the clause "Start a new post" (from the motivating example of Section 2) to the text component `''New post''`, as the similarity value with this element of the UI is 0.87. When the technique finds a clause to be semantically related to an element in the ontology, it creates an abstract step (Create-Generic-A-Step) and adds the step to the list of abstract steps (line 19). Generic abstract steps differ from the abstract steps created by function Create-A-Step; for these steps, the action to be performed will be determined dynamically, during the next phase of the technique, by introspecting the runtime properties of the UI element identified. Finally, the algorithm terminates by returning the list of abstract steps (line 20).

3.3 UI Actions Search

This phase takes as input the abstract steps produced in the previous phase and a device on which to run the app and produces as output a concrete test that reproduces the relevant failure. It dynamically explores the relevant app looking for a sequence of UI actions that match the input abstract steps. Our technique generates test cases dynamically, rather than statically, because we found that the app navigation models computed by state-of-the-art static analysis tools are too imprecise and incomplete to be used in this context.

Algorithm 2 describes how Yakusu searches for test cases. The algorithm takes as input the list of abstract steps (*aSteps*), the relevant app (*ra*), and a test device (*td*). The output of the algorithm is a test case (*tc*), which is also the final output of the technique. At a high level, the algorithm explores the relevant app to find a mapping between abstract steps and UI actions to perform on specific elements of the UI. The state of the search is represented as a triple containing (1) the list of abstract steps not yet successfully processed, (2) the list of abstract steps already processed, and (3) the list of UI actions corresponding to the abstract steps processed so far. Lines 2 and 3 initialize the state. At line 2, the algorithm initializes *tc* with an empty list and *states* and *pStates* with an empty set. *states* denotes the set of states yet to be explored, while *pStates* denotes the set of states already explored. Line 3 assigns the initial search state $\langle aSteps, [], [] \rangle$ to set *states*. In the initial state, the list of remaining abstract steps corresponds to the list of abstract steps provided on input, whereas the list of processed abstract steps and its corresponding list of UI actions are empty.

After these initialization steps, the algorithm starts to execute its main loop, where each loop iteration processes one state (lines 4–69). The first step in the main loop selects the most promising state for the search (line 5) by calling Find-Best-State. This function selects the state with the highest number of successfully processed abstract actions, choosing one state randomly in case of ties, returns it, and removes it from set *states*. This depth-first search is often faster for mobile apps, given the cost for restarting the app in alternative search strategies [8]. After selecting the state to be processed, the algorithm starts the relevant app on the test device (Start) and restores the state of the app by running the list of UI actions associated with the list of already-processed abstract steps (Restore). It then calls function Get-Remaining-A-Steps to extract the list of remaining abstract steps to process on the given state

Algorithm 2: UI actions search.

```
   Input : aSteps: list of abstract steps describing how to reproduce the relevant failure
           ra: relevant app
           td: test device
   Output: tc: test case that reproduces the bug report
1  begin
2      tc = [], states = ∅, pStates = ∅
3      states.Add(State(aSteps, [], []))
4      while states ≠ ∅ do
5          state = Find-Best-State(states)
6          Start(td, ra)
7          Restore(td, ra, state)
8          rASteps = state.Get-Remaining-A-Steps()
9          while rASteps ≠ [] do
10             aStep = rASteps.Remove(0)
11             //Case 1: Abstract steps without a UI element bound to them
12             if aStep == AStep ∧ ¬aStep.Has-Element() then
13                 action = aStep.Get-Action()
14                 target = aStep.Get-Target()
15                 ps = aStep.Get-Props()
16                 elements = Find-UI-Element(td,ra,action,target,ps)
17                 if elements ≠ [] then
18                     element =elements.Remove(0)
19                     foreach aElement ∈ elements do
20                         cstate = state.Copy()
21                         caStep = aStep.Copy()
22                         crASteps = rASteps.Copy()
23                         caStep.Set-Element(aElement)
24                         crASteps.Add(0, caStep)
25                         cstate.Set-Remaining-A-Steps(crASteps)
26                         states.Add(cstate)
27                     aStep.Set-Element(element)
28                     rASteps.Add(0, aStep)
29                     continue
30                 else
31                     rASteps.Add(0, aStep)
32                     if aStep.Get-Random-Count() < α then
33                         rUIAction = Random-UI-Action(td, ra)
34                         state.Add-UI-Action(rUIAction)
35                         Perform(td, ra, rUIAction)
36                         continue
37                     else
38                         break
39             //Case 2: Abstract steps with a UI element bound to them
40             else if aStep == AStep ∧ aStep.Has-Element() then
41                 element = aStep.Get-Element()
42                 if From-Heuristic(element) then
43                     hUIAction = Heuristic-UI-Action(element)
44                     state.Add-UI-Action(hUIAction)
45                     Perform(td, ra, hUIAction)
46                 action = aStep.Get-Action()
47                 if action == null then
48                     action = Find-Action(td, ra, element)
49                 ps = aStep.Get-Props()
50                 nUIAction = UI-Action(action, element, ps)
51                 state.Add-UI-Action(nUIAction)
52                 state.Get-Processed-A-Steps().Add(aStep)
53                 Perform(td, ra, nUIAction)
54                 continue
55             //Case 3: Generic abstract steps
56             else if aStep == GenericAStep then
57                 sstate = state.Copy()
58                 srASteps = rASteps.Copy()
59                 sstate.Set-Remaining-A-Steps(srASteps)
60                 states.Add(sstate)
61                 target = aStep.Get-Target()
62                 nAStep =Create-A-Step(null, target, [])
63                 rASteps.Add(0, nAStep)
64                 continue
65         if rASteps == [] then
66             tc = Generate-Test-Case(state.Get-UI-Actions())
67             return tc
68         else
69             pStates.Add(state)
70     state = Find-Best-State(pStates)
71     tc = Generate-Test-Case(state.Get-UI-Actions())
72     return tc
```

(*rASteps*). The inner loop (lines 9–64) processes these abstract steps. In the following, we refer to an abstract step whose action (e.g., click, rotate, scroll) has not been determined as a *generic abstract step*. For example, the first step in the bug report from Figure 1, "Start a new post", is a case of generic abstract step. Each iteration of the inner loop handles one of the following three types of abstract steps: abstract steps without a UI element bound to them, abstract steps with a UI element bound to them, and generic abstract steps.

Case 1: Abstract steps without a UI element bound to them. In this first case (lines 12–38), the algorithm must first find a UI element that matches the target specified by the abstract step and then perform the corresponding action on it. To do so, it first extracts the action (Get-Action), the target (Get-Target), and the properties (Get-Props) from the abstract step. It then looks for a potentially matching UI element (Find-UI-Element) by processing the properties of elements currently visible in the UI of the relevant app. Specifically, function Find-UI-Element compares the textual content of the target with the properties of UI visible elements using word embeddings computed, also in this case, from a word2vec model [32, 33]. The comparison is based on the same types of UI element properties extracted to compute the ontology, namely, label, icon, and identifier. A UI element is considered a match for the target if the cosine similarity between one of the element's properties vector representation and the target's vector representation is above 0.5. The computation of vector representations and their comparison follows the same approach described in Section 3.2. In the example from Section 2, the technique would find that the button with label "PUBLISH", appearing at the top-right corner in Figure 3, is a match for target "Publish" of the abstract step generated for the sentence "Tap on the Publish button", as their vector representations have cosine similarity 1.0 (they are the same word). Function Find-UI-Element uses two heuristics to also "reveal" elements within the screen of the relevant app that may not be readily visible. These heuristics try to identify this hidden UI elements by opening the menus and scrolling through the lists, respectively, in the current screen of the app.

At this point (line 16), *elements* stores (in descending order of cosine similarity value) the set of potential UI element candidates returned by Find-UI-Element. If this set is empty (lines 30–38), the search for candidates was unsuccessful, which typically happens when a step was missing in the bug report. In that case, the algorithm generates a random UI action (*rUIAction*) and continues to the next iteration, trying to fill the gap in the report, unless the number of random UI actions generated for the current abstract step exceeded a predefined threshold (α). If the search is successful (lines 17–29), set *elements* is non empty. Yakusu extracts from this set the element with highest cosine similarity, assigns the element to the abstract step, and reprocesses the abstract step with this UI element bound to it as discussed in the next paragraph (Case 2). For the other elements in the set, the algorithm conceptually forks the execution (lines 19–26) by copying the search state and setting the top remaining abstract step (*aStep*) to have its UI element (*aElement*) adjusted accordingly. These copied states will be processed if the technique does not successfully process all abstract steps in the current execution.

Case 2: Abstract steps with a UI element bound to them. In this second case, the algorithm performs a UI action on the element bound to the abstract step (lines 40–54). To do so, the technique either extracts the action associated with the abstract step, or identifies one if there is no associated action (which is possible after a generic abstract step has been processed, as discussed in Case 3). At this point, the technique saves the action in the form of a UI action in the state, adds the abstract step to the list of satisfied steps associated with the state, executes the UI action on the relevant app, and moves forward to analyze the next abstract steps that needs to be processed (line 54).

Case 3: Generic abstract steps. In this third and last case (lines 56–64), the algorithm processes steps that may be related to some UI element in the relevant app, but do not specify any action to be performed on such element. These steps typically either correspond to vaguely expressed actions or are not actual steps. The algorithm accounts for these cases by virtually forking the execution of the relevant app: one execution tries to perform the generic abstract step; the other simply skips this step. To do so, Yakusu adds to the list of current states a copy of the state (*sstate*) in which the generic abstract step is removed from the list of steps to be processed (line 60). Then, Yakusu continues the exploration for the current state by generating an abstract step whose target is the one associated with the generic abstract step and whose action is undefined. Then, the technique adds the abstract step to the beginning of the list of steps to be processed and moves forward to process the generic abstract step as an abstract step (line 64).

When the technique finishes processing the abstract steps in a state, it checks the content of the list of remaining abstract steps (lines 65–69). If the list is empty, the search process successfully terminates returning a test case with the list of UI actions associated with the current state (Generate-Test-Case). In case the list is not empty, Yakusu adds the state to the list of processed states (line 69) and continues. When the technique is not able to successfully process all abstract steps in any of the states analyzed (line 70), it finds the state that satisfied the highest number of abstract steps (Find-Best-State) and generates a "partial" test case that consists of the UI actions associated with that state.

4 IMPLEMENTATION

We implemented Yakusu in a tool that is publicly available [13] and consists of three main modules. The *ontology extraction module* is written in Java and uses Gator [42, 52] to analyze the source code of the app and build an ontology, as well as encoding UI elements and their properties. The ontology is stored in JSON format [24]. The *bug report analysis* module is a standalone Java program and uses the Stanford CoreNLP framework [30] to translate the text of the report into dependency trees and process them. The computation of word embeddings using word2vec is implemented in Python as a web service that leverages the Gensim library [50] and the dataset based on Google News [19]. Generated abstract steps are also stored in JSON format. The *executable actions search* module is built on top of the Espresso framework [18]. Using this framework, the implementation of Algorithm 2 is able to introspect the UI content of the app and perform UI actions. Finally, generated test cases follow the format specified by the Espresso framework, and their source code is generated using the JavaPoet library [45].

5 EVALUATION

This section discusses our empirical evaluation. To assess the expressiveness and efficiency of Yakusu, we investigated the following research questions:

- **RQ1 (effectiveness):** Can Yakusu translate bug reports written in natural language into executable test cases?
- **RQ2 (efficiency):** What is the cost of running Yakusu?

5.1 Experimental Benchmarks and Setup

We used a set of bug reports from real-world apps to evaluate Yakusu. As the use of Espresso to encode test cases requires the source code of the app to be available, we focused on apps from GitHub [14]. We queried the GitHub database for issues using keywords "android", "crash", "reproduce", and "version" and considered only issues submitted after January 1st, 2017. We used keyword "android" to find issues that relate to Android apps; we included keyword "crash" to find issues that could be easily verified; we used keyword "reproduce" because we were interested in bug reports that describe how to replicate a bug; finally, we included keyword "version" to make sure that the specific version of the app and operating system involved in the issue were available. We considered only issues created after January 1st 2017 to avoid considering apps that could have outdated dependencies.

This search returned 2709 issues, of which we randomly selected 100 for further processing. As a side note, we found that 79 of these 100 issues had been created by individuals who did not perform any commit to the repository, indicating that they were users not involved in the development of the corresponding app.

To be able to answer RQ1 in an accurate way, we first had to make sure that the issues considered were indeed reproducible. To that end, for each of the 100 issues selected, we first tried to build the version of the app specified in the report. In case a version was missing, we used the the most recent working commit before the date of the report to build the app. Somehow surprisingly, we found that building and setting up apps can be fairly complex and time consuming. In some cases, for example, we had to update outdated dependencies or set up server components for an app to compile and run. Overall, we could build and setup 91 of the 100 apps associated with the set of bug reports considered. For the remaining 9 apps, we either encountered compilation errors that we could not fix (7 apps) or could not find the code associated with the issue because the repository was initially used only for bug reporting (2 apps).

We then tried to manually reproduce the issues reported for the 91 apps that we were able to build and were successful for 62 of them. There were several reasons why we could not reproduce the issues involving the remaining 29 apps. In 9 cases, the report contained only a stack trace, and this trace did not provide us with enough information to recreate the issue described in the report. In 7 cases, the app interacted with a remote server outside of our control, and the communication protocol between app and server had changed, preventing the app from running. In 3 cases, the issue depended on a specific hardware/software configuration that was not available to us. In the last case, part of the issue description in the report was written in a language other than English, preventing us from fully understanding the issue.

5.2 Results

5.2.1 RQ1 (Effectiveness). To answer RQ1, we applied our technique to the set of 62 reproducible issues discussed above. Overall, Yakusu was able to successfully generate a test case for 37 of these 62 issues. (a success rate of 59.7%). In order to consider a generated test successful, we manually checked whether (1) the actions in the generated test matched the steps that a human would perform and (2) the stack trace generated by the test case corresponded to the one in the bug report (if one was present). Table 1 reports results for 25 of the cases, as 12 of the 37 issues simply required to launch the app and were trivial to reproduce. Details on the complete list of analyzed issues are publicly available [13]. It is worth mentioning that certain apps required some setup (e.g., user authentication) before they could be tested. As it is typical in these cases, and during testing in general, in our evaluation we provided this once-per-app setup information to Yakusu.

On Table 1, the columns under header *Benchmark* provide, for each app/issue considered, the identifier of the issue (*ID*), the name of the app (*Name*), the GitHub issue number (*Issue*), the lines of code in the app (*LOC (K)*), and the number of stars for the app on GitHub (*Stars*), which is a measure of popularity. Issues are ordered by their identifier numbers, whose values correspond to the order in which we (randomly) selected them.

The columns labeled *Actions* show the number of UI actions required to manually reproduce the issue described in the bug report, which consist of the sum of the number of actions that are explicitly (A_e) and implicitly (A_i) documented in the bug report. These latter are actions that were not specified in the bug report but that need to be performed to reproduce the issue. As the table shows, for 9 of the 25 issues (rows with A_i>0), at least one implicit action was involved in the reproduction task. In particular, for MifosX, the number of implicit actions was higher than number of explicit actions. (The implicit actions, in this case are "opening a sliding menu", "clicking on any element of a list", and "clicking on the element of the UI displaying an icon".) It is important to stress that the presence of implicit actions is far from rare, and generating tests for bug reports that involve such actions is particularly challenging. Therefore, the fact that Yakusu was able to suitably handle these cases indicates its effectiveness and potential usefulness.

The columns labeled *Steps* describe the number of abstract steps generated by the technique. Specifically, they show the number of abstract steps (*AS*) and the number of generic abstract steps (AS_g) generated by Yakusu for each issue. The sum of these two numbers, *AS* and AS_g, corresponds to the number of abstract steps provided as input to the search phase of the technique, which is responsible for generating an actual test case. This sum can be different from the sum of actions for two reasons. First, implicit actions are not translated into abstract steps, as they are not present in the bug report. Second, there could be generic abstract steps that might not actually describe a UI action. As an example, consider the bug report associated with the Open Event issue, which includes the following sentence: "Take the pull of the latest code". Yakusu translates this sentence into a generic abstract step that does not describe a specific action on the UI. As explained in Section 3, our technique handles this situation by forking an execution that discards this step. Conversely, for seven other issues (i.e., issues 01, 35, 40, 51, 84, 97, and 99), generic abstract steps are essential to reproduce the issue. This number suggests that Yakusu is able to handle actions in bug reports that are expressed at different levels of abstraction. It is also worth noting that seven of the bug reports reproduced by Yakusu did not contain header "reproduce", which provides initial evidence that Yakusu is able to handle less structured bug reports.

The columns labeled *Search* provide information on the outcome of the UI-actions-search phase of the technique. Specifically, they show statistics of the search that led to the generation of the test case corresponding to the input abstract steps: number of states generated (S_g), number of states processed (S_p), number of heuristics used (H), and number of random UI actions in the generated test case (R). In 16 cases, the search never had to select a different state to explore other than the current state (entries with S_g>1 and S_p=1). These are cases of quick successful runs, in which the technique did not have to start the app again and restore its state (lines 6-7 from Algorithm 2). In another 6 cases, conversely, the technique had to explore other states to find the list of UI actions that satisfied all the input abstract steps (entries with S_g>1 and S_p>1); these cases highlight the importance of tracking and exploring multiple states during the search. As columns H and R show, in creating test cases, Yakusu used at least one of its heuristics and generated at least one random action in seven and six cases, respectively.

Finally, the columns labeled *Tests* show the number of required (TC_s) and non-required (TC_a) statements in the generated test cases. As the table shows, Yakusu generated non-required statements in only four cases.

False negatives. We investigated the 25 (i.e., 62-37) cases in which Yakusu was unable to successfully generate a test case and grouped them into four categories:

Category 1 (7 cases): Reports with actions that need to be performed outside of the app (i.e., actions on the UI of the operating system). To address these cases, we could extend the ontology generated by Yakusu with system actions.

Category 2 (3 cases): Reports with actions on UI elements whose properties cannot be introspected at runtime (e.g., custom views [17]). We plan to explore how to make such properties available at runtime through app instrumentation.

Category 3 (11 cases): Reports in which a single step corresponds to multiple implicit UI actions. The sentence "After a while browsing through folders app crash [sic] immediately" [6], for instance, should be translated to a sequence of clicks on various folders, but Yakusu fails to interpret the sentence correctly. We plan to investigate ways to handle these cases by leveraging related work (e.g., [9, 36, 37, 43]) and incorporating additional domain knowledge, possibly on demand.

Category 4 (4 cases): Reports with actions that involve multi-touch gestures (e.g., pinch zoom). Yakusu could handle these actions through suitable, albeit extensive, engineering.

False positives. We did not observe false positives, that is, successfully generated test cases that did not lead to the failure described in the bug report. If these cases were to occur, in the context of bug reports describing crashes, Yakusu could address this issue by filtering out non-crashing tests.

Table 1: Benchmarks for which YAKUSU could generate a test case reproducing the bug report. For each benchmark considered: *ID* = identifier; *Name* = name; *Issue* = identifier of the GitHub issue considered; *LOC(K)* = # lines of code (thousands); *Stars* = # stars on GitHub; A_e = # explicit actions in the issue; A_i = # implicit actions in the issue; *AS* = # abstract steps generated; AS_g = # generic abstract steps generated; S_g = # states generated; S_p = # states processed; *H* = # heuristics used in the successful state; *R* = # random actions generated in the successful state; TC_s = number of statements in the test; TC_a = number of additional statements in the test; T_{oe} = time taken by the ontology extraction phase; T_{bra} = time taken by the bug report analysis phase; T_{uias} = time taken by the UI actions search phase.

Benchmarks					*Actions*		*Steps*		*Search*				*Tests*		*Cost*		
ID	*Name*	*Issue*	*LOC (K)*	*Stars*	A_e	A_i	*AS*	AS_g	S_g	S_p	*H*	*R*	TC_s	TC_a	T_{oe}	T_{bra}	T_{uias}
01	TACHIYOMI	880	38	1603	4	1	-	4	10	1	1	-	5	-	54s	18s	1m12s
03	TWIDERE	738	141	1672	2	-	2	-	12	1	-	-	2	-	2m34s	15s	1m07s
05	SIGNAL	6660	125	9803	1	-	1	-	2	1	-	-	1	-	1m23s	17s	6m57s
08	REDREADER	516	42	830	2	-	2	2	5	1	-	-	4	2	25s	19s	1m25s
14	SILENCE	557	109	871	1	-	1	-	2	1	-	-	1	-	43s	15s	5m08s
23	K-9 MAIL	1910	136	3868	3	-	3	-	8	1	-	-	3	-	32s	16s	1m00s
25	NEXTCLOUD	883	78	768	1	2	1	-	1	1	1	1	3	-	34s	16s	2m08s
27	BUTTERKNIFE	46	5	165	1	-	1	-	3	1	-	-	1	-	13s	16s	09s
35	ODK COLLECT	360	49	292	2	-	1	4	11	3	-	-	2	-	30s	18s	3m51s
39	PIX-ART MESSENGER	127	58	30	3	-	3	-	2	1	-	-	3	-	43s	15s	31s
40	YALP STORE	204	17	960	5	-	4	1	11	1	-	-	5	-	16s	17s	2m25s
50	OCREADER	48	13	49	3	-	3	-	8	2	-	-	3	-	22s	17s	1m33s
51	WORDPRESS	5497	180	1758	3	-	2	1	2	1	-	-	3	-	1m35s	16s	3m18s
52	SIGNAL	6924	126	9803	2	1	2	-	3	1	-	4	6	3	1m16s	17s	18m10s
57	OPEN EVENT	1402	18	344	1	1	1	1	3	2	1	-	2	-	36s	20s	2m48s
68	TAGMO	12	32	663	1	1	1	1	10	1	-	3	4	2	17s	16s	31s
69	ANKIDROID	4586	101	1231	5	-	5	-	8	1	-	-	5	-	29s	17s	28m55s
73	K-9 MAIL	2612	137	3868	2	-	2	-	6	3	-	-	2	-	35s	18s	3m22s
74	CLUTTR	2	13	9	2	-	2	-	6	1	-	-	2	-	26s	15s	24s
78	NEXTCLOUD	850	74	768	2	1	2	-	1	1	1	-	3	-	32s	15s	2m20s
84	K-9 MAIL	2019	136	3868	1	-	-	1	2	1	-	-	1	-	32s	17s	45s
95	MIFOSX	734	65	85	2	3	2	-	3	1	1	2	5	-	39s	18s	1m25s
96	SCREENRECORDER	25	7	62	3	2	3	-	1	1	2	-	5	-	15s	17s	54s
97	NEXTCLOUD	1061	81	768	1	1	-	2	4	2	1	-	2	-	37s	23s	9m28s
99	FLASHCARDS	13	5	8	3	-	2	1	6	2	-	3	4	1	15s	18s	58s

In summary, we consider the results for RQ1 encouraging. Despite the limitations presented above, which can be addressed as discussed, YAKUSU was already able to generate test cases for a majority of bug reports while generating no spurious tests.

5.2.2 RQ2 (Efficiency). To answer RQ2, we measured the time taken to run each phase of the technique on a MacBook Pro with 2.8 GHz i7 processor and 16GB of RAM. Table 1 shows, for each issue considered, the time required to extract the ontology (T_{oe}), perform bug report analysis (T_{bra}), and explore the app to generate a test case (T_{uias}). The times in the table are expressed in minutes (m) and seconds (s).

As the table shows, the UI-actions-search phase is where YAKUSU spent most of its time, followed by the ontology-extraction phase, and then the bug-report-analysis phase. Our technique generates a test case in less than five minutes overall in most cases, with the average and median times being 5m00s and 2m58s, respectively. In only two cases, the execution time was above ten minutes. In particular, ANKIDROID (issue 69) is the case in which YAKUSU takes the longest time to generate a test case: 29m41s. The execution time for this benchmark is dominated by the UI actions search phase of the technique, which is 28m55s. This value is higher than the one associated with other benchmarks due to a higher number of computations of the cosine similarity value. (This operation is relatively expensive compared to other operations because it requires to issue a network request from the test device to get the similarity value, as presented in Section 4.) Considering that the execution time is fairly low even in the worst case, we did not further investigate this issue, nor tried hard to optimize YAKUSU. In fact, these execution times suggest that YAKUSU could be used to monitor bug reports and generate test cases throughout the day, as opposed to overnight only. Moreover, for issues that cannot be translated successfully and may result in longer running explorations of the app states, developers could provide a suitable timeout.

5.3 Threats To Validity

As it is the case for most empirical evaluations, there are both external and construct threats to validity associated to the results we presented. In terms of external validity, our results might not generalize to other bug reports or apps. In particular, we only considered 100 bug reports. This limitation is an artifact of the complexity involved in manually building and setting up the infrastructure to run the app associated with a bug report. To mitigate this threat, we used randomly selected real-world bug reports from different apps. An additional threat could be posed by the fact that we used only open source apps in the evaluation. However, the evaluation includes apps such as K-9 MAIL, SIGNAL, and WORDPRESS, which have complex functionality, hundreds of widgets, and millions of users. We believe that, given the complexity of the apps we analyzed, YAKUSU should also be applicable to other types of apps. In terms of construct validity, there might be errors in the implementation of our technique. To mitigate this threat, we extensively inspected the results of the evaluation manually.

6 RELATED WORK

The problem of synthesizing code from natural language descriptions was studied in different domains (e.g., [5, 7, 10, 12, 16, 21, 26–28, 31, 41, 47, 49]). In the following, we discuss the work most closely related to our technique.

Branavan and colleagues [5] and Lau and colleagues [27] independently used NLP techniques to extract concrete actions from text documents for improving productivity of various tasks (e.g., troubleshooting, learning from tutorials). Branavan and colleagues [5] proposed a reinforcement learning approach for translating tutorials of desktop applications into a list of concrete actions. Their approach uses a learning algorithm, trained with a corpus of text documents, to infer actions. Yakusu uses dependency parsing to infer actions from bug reports because, in our application domain, a set of documents (i.e., bug reports) is not always available for training. Lau and colleagues [27] proposed an automated approach to translate into test cases documents that describe how to accomplish various tasks on the web. Their work and ours both translate natural language sentences into concrete actions. However, their approach assumes that instructions are properly segmented (i.e., one action per sentence), while Yakusu is able to automatically identify segments (referred to as clauses in Section 3.2) in a sentence. In a sense, Branavan and colleagues and Lau and colleagues realized that the problem of interpreting arbitrary hand-written documents, although very challenging in general, becomes more amenable to machine processing when the input language is restricted to a certain domain. Yakusu builds on this idea, while weakening the assumptions on the input made by the aforementioned techniques.

Thummalapenta and colleagues [47] proposed ATA, an approach to translate web application tests, written in natural language by professional testers, into their corresponding scripts. Both Yakusu and ATA identify actions using dependency parsing and account for the fact that multiple choices for a target might be present during test generation. However, there are generally considerable differences between professionally written test specifications and bug reports, which prevents ATA from being straightforwardly applied to bug reports. For instance, bug reports written by users do not generally follow a precise structure and can use different levels of abstractions in describing the steps to reproduce an issue. Our work takes these aspects into account by mapping the description of the bug report to an ontology of the app. Additionally, our technique is also able to handle cases in which some necessary steps of the test are not explicitly mentioned in the bug report.

Le and colleagues [28] proposed SmartSynth, a technique that combines NLP and program synthesis to produce automation scripts. For example, SmartSynth could handle an automation script like the following one: "When I receive a new SMS message, if the phone is connected to my car's bluetooth, it reads out loud the message content and replies the sender 'I'm driving'." SmartSynth and Yakusu differs in terms of their application context, as SmartSynth generates system events (e.g., "Turn off GPS") as opposed to UI events. In this sense, SmartSynth can be seen as complementary to Yakusu, which could leverage system events to increase the range of bug reports it can handle (e.g., "Turn off GPS before clicking button x").

More broadly, our work relates to the area of field failure reproduction (e.g., [3, 22, 23, 38, 39, 46, 53]). We believe that Yakusu and these alternative techniques tend to have complementary advantages and disadvantages. In particular, techniques for failure reproduction based on partial information (e.g., stack traces, partial event sequences) do not require NLP but tend to involve heavy-duty analysis. Overall, these techniques and Yakusu are mostly orthogonal, and a developer may decide which one to use for a specific bug report based on the information available or just use multiple ones and see which one performs best.

7 CONCLUSION

When processing a bug report, developers would typically try to reproduce the failure described in the report, so as to be able to investigate it and identify its causes. This is a time consuming and challenging task, due to the typically large number of bug reports and the effort involved in analyzing them. To help developers with this task, we proposed Yakusu, a technique that combines natural language processing and program analysis to extract from a bug report a test case that reproduces the issue described in the report.

We implemented and empirically evaluated Yakusu by running it on a set 62 real-word bug reports. Yakusu was able to automatically generate tests for 59.7% of the reports, which we believe is an initial, yet strong indication of the effectiveness of the approach; for all these reports the developers could simply use the test cases automatically generated by Yakusu instead of having to study the reports and understand how to recreate the issue described therein. It is also worth noting that this number could increase even further after we address some of the existing limitations of the approach.

We foresee a number of venues for future work. First, we will perform a user study in which we will assess the usefulness of our technique and to what extent developers can benefit from it. Second, we will investigate ways to extend the ontology extracted from an app by analyzing callbacks, that is, the functions that are executed as a consequence of UI actions; the name of a callback function or the data elements used therein could help us generate a better characterization of the UI event associated to that function through a callback. Third, we will extend the technique to include system actions and events, so as to be able to reproduce issues that involve operations performed outside the specific app of interest. To do this, we will study ways to build an ontology for relevant system elements, actions, and events. Fourth, we will investigate ways to incorporate more domain knowledge to be able to handle descriptions of (macro) steps in a report that should be interpreted as sequences of lower-level actions on the UI (e.g., "enter credential" being interpreted as providing a login and a password in some form). Fifth, we will study ways to interpret the description of (non-crashing) failures in bug reports, so as to be able to generate oracles for the automatically generated tests. Although this is not needed for crashing bugs, it can make the technique useful for more subtle failures. Sixth, we will investigate ways to adapt the technique to web and GUI-based desktop apps. Seventh, we will explore ways to extend Yakusu so that it can generate test cases from specifications in written or spoken English. We will also consider ways to use our approach for identifying duplicate bug reports. Finally, and more on the engineering side, we will extend the implementation of the technique to support custom views and multi-touch gestures, which will allow us to handle reports that involve these kinds of events.

ACKNOWLEDGMENTS

Jacob Eisenstein provided useful input and feedback on the NLP part of our technique. This work was partially supported by NSF under awards CCF-1161821 and CCF-1563991.

REFERENCES

[1] 0nko. 2017. Notification icon: App crash when publishing a post. Retrieved June 8, 2018 from https://github.com/wordpress-mobile/WordPress-Android/issues/5497

[2] Gabor Angeli, Melvin Jose Johnson Premkumar, and Christopher D. Manning. 2015. Leveraging Linguistic Structure For Open Domain Information Extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*. The Association for Computer Linguistics, Stroudsburg, PA, USA, 344–354.

[3] Shay Artzi, Sunghun Kim, and Michael D. Ernst. 2008. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *22nd European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 542–565.

[4] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 308–318.

[5] S. R. K. Branavan, Harr Chen, Luke S. Zettlemoyer, and Regina Barzilay. 2009. Reinforcement Learning for Mapping Instructions to Actions. In *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. The Association for Computer Linguistics, Stroudsburg, PA, USA, 82–90.

[6] Bullnados. 2017. Main and Nightly Version crashing all time on LG G4. Retrieved June 8, 2018 from https://github.com/nextcloud/android/issues/760

[7] Cohan Sujay Carlos. 2011. Natural Language Programming Using Class Sequential Rules. In *Proceedings of the Fifth International Joint Conference on Natural Language Processing*. The Association for Computer Linguistics, Stroudsburg, PA, USA, 237–245.

[8] 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*. ACM, New York, NY, USA, 623–640.

[9] Pedro Costa, Ana C. R. Paiva, and Miguel Nabuco. 2014. Pattern Based GUI Testing for Mobile Applications. In *2014 9th International Conference on the Quality of Information and Communications Technology*. IEEE Computer Society, Washington, DC, USA, 66–74.

[10] Anthony Cozzie, Murph Finnicum, and Samuel T. King. 2011. Macho: Programming with Man Pages. In *13th Workshop on Hot Topics in Operating Systems*. USENIX Association, Napa, CA, United States.

[11] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation*. European Language Resources Association (ELRA), Genoa, Italy, 449–454.

[12] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R., and Subhajit Roy. 2016. Program Synthesis Using Natural Language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, New York, NY, USA, 345–356.

[13] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Yakusu. Retrieved June 8, 2018 from http://www.cc.gatech.edu/~orso/software/yakusu

[14] GitHub 2018. GitHub. Retrieved June 8, 2018 from https://github.com

[15] GitHub 2018. Manually creating a single issue template for your repository. Retrieved June 8, 2018 from https://help.github.com/articles/manually-creating-a-single-issue-template-for-your-repository

[16] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 213–224.

[17] Google 2018. Custom View Components. Retrieved June 8, 2018 from https://developer.android.com/training/custom-views/index.html

[18] Google 2018. Espresso. Retrieved June 8, 2018 from https://developer.android.com/training/testing/espresso/index.html

[19] Google 2018. Google News Vectors Negative 300. Retrieved June 8, 2018 from https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM

[20] Google 2018. Reporting Bugs. Retrieved June 8, 2018 from https://source.android.com/setup/report-bugs

[21] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 416–432.

[22] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 474–484.

[23] Wei Jin and Alessandro Orso. 2013. F3: Fault Localization for Field Failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 213–223.

[24] JSON 2018. Introducing JSON. Retrieved June 8, 2018 from https://www.json.org

[25] Dan Jurafsky and James H Martin. 2014. *Speech and language processing*. Pearson Education, London, UK.

[26] Mathias Landhäußer, Sebastian Weigelt, and Walter F. Tichy. 2017. NLCI: a Natural Language Command Interpreter. *Automated Software Engineering* 24 (2017), 839–861.

[27] Tessa A. Lau, Clemens Drews, and Jeffrey Nichols. 2009. Interpreting Written How-To Instructions. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. AAAI Press, Bellevue, Washington, USA, 1433–1438.

[28] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, 193–206.

[29] Dennis Lee. 2016. How to write a bug report that will make your engineers love you. Retrieved June 8, 2018 from https://testlio.com/blog/the-ideal-bug-report

[30] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*. The Association for Computer Linguistics, Stroudsburg, PA, USA, 55–60.

[31] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. 2013. Integrating Programming by Example and Natural Language Programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. AAAI Press, Bellevue, Washington, USA, 661–667.

[32] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013).

[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013*. Curran Associates Inc., Lake Tahoe, Nevada, USA, 3111–3119.

[34] Gary Miner, John Elder, Thomas Hill, Robert Nisbet, Dursun Delen, and Andrew Fast. 2012. *Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications*. Academic Press, Orlando, FL, USA.

[35] Chris Moody. 2015. A Word is Worth a Thousand Vectors. Retrieved June 8, 2018 from https://multithreaded.stitchfix.com/blog/2015/03/11/word-is-worth-a-thousand-vectors

[36] Rodrigo M. L. M. Moreira, Ana C. R. Paiva, and Atif Memon. 2013. A Pattern-Based Approach for GUI Modeling and Testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Washington, DC, USA, 288–297.

[37] Ines Coimbra Morgado and Ana C. R. Paiva. 2015. Testing Approach for Mobile Applications through Reverse Engineering of UI Patterns. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE Computer Society, Washington, DC, USA, 42–49.

[38] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *32nd International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 284–295.

[39] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson. 2015. JCHARMING: A Bug Reproduction Approach Using Crash Traces and Directed Model Checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, Washington, DC, USA, 101–110.

[40] Jason Ostrander. 2012. *Android UI Fundamentals: Develop and Design*. Peachpit Press, Berkeley, CA, USA.

[41] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, New York, NY, USA, 357–367.

[42] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, New York, NY, USA, 143–153.

[43] Clara Sacramento and Ana C. R. Paiva. 2014. Web Application Model Generation through Reverse Engineering and UI Pattern Inferring. In *2014 9th International Conference on the Quality of Information and Communications Technology*. IEEE Computer Society, Washington, DC, USA, 105–115.

[44] John Saito. 2016. Making a case for letter case. Retrieved June 8, 2018 from https://medium.com/@jsaito/making-a-case-for-letter-case-19d09f653c98

[45] Square 2018. JavaPoet. Retrieved June 8, 2018 from https://github.com/square/javapoet

[46] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. 2000. jRapture: A Capture/Replay Tool for Observation-based Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 158–167.

[47] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. 2012. Automating Test Automation. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 881–891.

[48] Universal Dependencies 2018. Universal Dependencies. Retrieved June 8, 2018 from http://universaldependencies.org

[49] David Vadas and James R. Curran. 2005. Programming With Unrestricted Natural Language. In *Proceedings of the Australasian Language Technology Workshop*. Australasian Language Technology Association, Sydney, Australia, 191–199.
[50] Radim Řehůřek. 2018. Gensim. Retrieved June 8, 2018 from https://radimrehurek.com/gensim
[51] WordPress 2018. WordPress. Retrieved June 8, 2018 from https://play.google.com/store/apps/details?id=org.wordpress.android
[52] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Washington, DC, USA, 658–668.
[53] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. 2014. MIMIC: Locating and Understanding Bugs by Analyzing Mimicked Executions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 815–826.