Repairing Fragile GUI Test Cases Using Word and Layout Embedding

Juyeon Yoon
KAIST
Daejeon, Republic of Korea
juyeon.yoon@kaist.ac.kr

Seungjoon Chung
KAIST
Daejeon, Republic of Korea
s.j.chung@kaist.ac.kr

Kihyuck Shin
Samsung Electronics
Seoul, Republic of Korea
kihyuck.shin@samsung.com

Jinhan Kim
KAIST
Daejeon, Republic of Korea
jinhankim@kaist.ac.kr

Shin Hong
Handong Global University
Pohang, Republic of Korea
hongshin@handong.edu

Shin Yoo
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

***Abstract*—Smartphone vendors apply both device and brand-specific customisations to the underlying operating systems, resulting in a wide range of device configurations. It is crucial that all of the device variations provide compatibility with the default version of the underlying operating system, such as Android. To ensure that widely and commonly used apps run on each of these device variations without any problem, vendors depend on automated GUI level testing of widely and commonly used apps: the failure of a GUI test script that emulates a routine usage of these apps would raise an alarm that a recent change made to a specific device variation may have caused a regression fault. These GUI level compatibility smoke tests are unique in the sense that they are GUI level automated test scripts that are written outside the software development lifecycle of the target apps: they are written and maintained by the engineers of the smartphone vendors, and not the app developers. As such, these test scripts are extra vulnerable to the fragility of GUI test scripts, which are already known to be fragile when maintained by app developers. This paper introduces a repair technique for View Identification Failures (VIFs) in those smoke tests so that the smartphone vendors can quickly update their GUI test scripts when they break due to changed view ids. Our technique matches view ids between old and new versions of the target app based on various similarity metrics such as the semantic embedding similarity between ids and GUI labels, and layout similarity based on node embeddings of the GUI layout tree. We evaluate the proposed technique using 512 VIFs collected from real-world Android mobile apps. The proposed technique can repair 72% of the 512 studied VIFs with only one attempt, compared to 28% repaired using lexical distance-based matching.**

***Index Terms*—GUI Testing, Test Repair, View Identification Failure, Android**

I. Introduction

Android mobile platform occupies more than 80% of the world-wide smartphone market [1] and has a rich and diverse ecosystem that includes various devices as well as vendor specific customisations both in hardware and software. While these customisations add value to customers, they also contribute to the phenomenon known as device fragmentation [2]: due to the numerous combinations of device models, vendor specific customisations, and rapidly evolving operating system versions, it is extremely difficult to test an Android mobile app exhaustively against all configurations.

While the device fragmentation presents testing challenges for developers of individual apps, it also poses a unique challenge for smartphone vendors. When developing a new hardware device, or a software customisation layer, smartphone vendors need to make sure that the newly developed variation is compatible with the app ecosystem, i.e., they should guarantee that important, widely used mobile apps can run without any problem on top of the new variation under development. A failure to execute common and popular apps can be considered as a regression from the perspective of the smartphone vendors.

One way to prevent such regression is to adopt automated GUI testing. A routine usage of target apps can be emulated with automated GUI test scripts, using GUI test automation framework such as Espresso.[1] Failures from these GUI smoke tests may suggest that recent changes made to the underlying device customisation contain some issues [**?**].

However, such an approach suffers from the same issue that plagues automated GUI testing of Android apps, which is the inherent fragility of GUI test scripts. A GUI test script is fragile because a relatively small change in the appearance of an app can break test cases that traverse the GUI [3]. One example of such fragility is View Identification Failures (VIFs) [4]. Typically, an automated GUI testing framework for Android first identifies a GUI element (i.e., a *view*), such as a button, and subsequently triggers a GUI event to the specified view in order to emulate user interaction. For example, the following is a test snippet written using Espresso: it finds a button with view identifier `btnConfirm` and clicks it.

```
onView(withId(R.id.btnConfirm))
  .perform(click());
```

A view identification failure occurs when the latest modification to the app source code changes the view identifier `btnConfirm` to `btnOk`, but the test case is still looking for a

[1] https://developer.android.com/training/testing/espresso

button with the old identifier. Consequently, the test execution terminates with a `ViewMatchingException`, or a compile error due to an unknown resource, despite the fact that the logic of both the app and the test case has not changed. VIF is listed as one of the major causes of Android test fragility in a recent survey of Android test practices [4]. The vendor written GUI smoke test scripts are even more VIF prone than GUI test scripts written by app developers, since they are separated from the development activities of the target apps: any changes in GUI are not directly visible or trackable for the vendor.

This paper presents VIFER, a **v**iew **i**dentification **f**ailure of t**e**st **r**epairer that exploits the similarities between old and new view elements in Android GUI. To repair a VIF, we need to match the old, obsolete view identifier to its new counterpart, and update it. To match an old view identifier to the new one, we exploit not only lexical and semantic distance between view identifiers themselves, but also compare the similarity between the relative positioning of the views in the overall GUI layout using graph node embedding. Using these distance metrics as features, we formulate the repair problem as a classification problem: given a pre-change identifier (used by the broken test case) and a set of post-change identifiers (collected in the modified app source code), we classify which of the post-change identifiers is most likely to match the pre-change identifier. The results of classification are presented as a ranking of post-change identifiers. An empirical evaluation based on real world VIFs shows that VIFER can repair 88.5% of the failures within three patching attempts, and precisely matches post-change identifiers for 71.7% of the studied failures by ranking the correct one on top. This is a significant improvement over the existing developer support tool based on lexical similarity only, which can repair 43.2% of the failures within three attempts.

The remainder of the paper is organised as follows. Section II describes the view identification failure in more detail and introduces the formulation of our repair technique. Section III presents the experimental setup of the empirical evaluation, the results of which are reported in Section IV. Section V presents the threats to validity. Section VI describes the related work, and Section VII concludes.

II. Repairing VIF By Identifier Matching

This section defines View Identification Failure (VIF) and presents our formulation of VIF repair as a classification problem. Subsequently, it also introduces various features we use for the classification.

A. View Identification Failure

The View class is the base class for all Android GUI widgets. An automated GUI test case emulates user interaction by 1) identifying a specific view (such as a button), and 2) invoking a specific GUI event (such as click). For the first step, the test case typically uses unique properties of the view, such as the given view identifier. View Identification Failure (VIF) occurs when an automated GUI test case fails to specify a view to which it wants to invoke a GUI event. This inconsistency is often caused by the fact that the GUI, i.e., the external appearance of an Android app, may go through modifications that do not involve any change of the underlying logic of the app. If only the GUI part of the app is changed, and not the test case code, then VIF occurs.

This paper specifically focuses on VIFs that involve outdated View identifiers. All GUI widgets can be given identifiers in the layout resource files in XML format that defines the visual structure in an Android application: these identifiers are accessible from within the app source code via the automatically generated class named `R.java` with `R.id.*` object. Views can be targeted by test cases via their identifiers.

Let us use the terms pre- and post-change identifier to refer to the outdated identifier in the test code, and the new, modified identifier in the app code; we will hereafter also use id to stand for view identifiers for the sake of brevity. If the relationship between pre- and post-change ids are completely arbitrary, an automated repair would not be feasible.

The common cause of view identification failure, however, is the fact that the external appearance of the app (i.e., the GUI design) can evolve even when the internal logic of the app does not. While the separation of concern between GUI appearance and the code allows room for VIFs, it also provides a ground for automated repair: if the underlying app logic has not changed at all, or changed only slightly, we can expect that the new identifiers or names given to the updated GUI elements will be similar to the previous names. If, for example, the latest GUI modification was simply to fix typos in view identifiers, the old and the new ids will be lexically similar. Even if the change was more complex than a simple typo fix, as long as the core logic wrapped inside the GUI remains largely the same, we expect the old and the new identifiers to be semantically similar. This continuity in the app functionality allows us a few different hypotheses, which we will consider in the following sections.

B. VIF Repair as a Classification Problem

We formulate VIF repair as a classification problem. Given two versions of Application Under Test (AUT), namely the original version and the latest version, we denote them as A and A' respectively. Let T be the GUI test case before the GUI change that is now broken due to VIF. The pre-change view id, i, only appears in T. Finally, let I_c be the set of all candidate post-change view ids that appear in A': $I_c = \{i_c | i_c \in A'\}$.

VIF repair is essentially the problem of finding $i' \in A'$ such that i' is the modified version of i. Given a pair of pre- and post-change view ids, (i, i_c), we label the pair 1 if the pair is the correct match between the same GUI widget with modified view ids, and 0 otherwise. We train a binary classifier, C, of view id pairs based on this labelling, using actual view identifier changes collected from open source Android app repositories.

$$C(i, i_c) = \begin{cases} 1, & \text{if } i_c = i', \\ 0, & \text{otherwise.} \end{cases}$$

To repair *unseen* identifier changes, we use softmax probabilities for label 1 obtained by the classifier. More specifically, for all candidate ids in I_c, the softmax probabilities are used to *rank* potential match candidate, i_c. We can repair the test T that is broken by VIF using the best matched candidate as follows (or top-k matches):

$$T' = T[i \leftarrow i_c] \text{ s.t. } i_c = \underset{i_c \in I_c}{\operatorname{argmax}}\ C(i, i_c)$$

Ideally, the correct match, i', should be ranked at the top, and will be used to update T to T' by replacing i with i'.

C. Identifier Similarity-based Input Features

We now turn our attention to the input features for the classification model. Intuitively, we assume that pre- and post-change view ids will be still similar to each other, due to the continuity in the underlying functionality of the AUT. If the change in GUI is mostly cosmetic, the pre- and post-change view ids are likely to represent the GUI widgets that perform similar functionalities. This, in turn, suggests that the developer may use similar names to communicate the functionality behind the view to other human developers [**?**]. In this paper, we consider both lexical and semantic similarities as input features.

1) Lexical Similarity Feature: One common type of superficial GUI changes related to VIF is the systematic change of view id naming convention. For example, it is possible that the pre-change id did not include the GUI widget type (e.g., `Confirm`), and the post-change id was modified so that it follows a naming convention that requires the widget type to be explicitly reflected in the id (e.g., `buttonConfirm` or `btnConfirm`). With naming convention changes, it is likely that part of the pre-change id remains the same in the post-change id. Consequently, we expect the lexical similarity, captured by Levenshtein distance [5], to capture this relationship. Levenshtein distance is a specific type of edit distance between two strings, a, and b, which is formally defined as follows:

$$lev(a,b) = \begin{cases} |a|, & \text{if } |b| = 0, \\ |b|, & \text{if } |a| = 0, \\ lev(a[1:], b[1:]), & \text{if } a[0] == b[0], \\ 1 + \min \begin{cases} lev(a[1:], b), \\ lev(a, b[1:]), \\ lev(a[1:], b[1:])), \end{cases} & \text{otherwise.} \end{cases}$$

We hypothesise that the correctly matched pair, (i, i'), will have higher lexical similarity than other pairs, for view modifications such as typo fixes or application of new naming conventions. Note that lexical similarity is the basis of the current technique deployed to assist developers with VIFs, and therefore also is our baseline.

2) Semantic Similarity Feature: It is also possible that the change causing VIF is more semantic than the simple lexical addition of pre- or suffixes. If the change made to the view id reflects a minor change in the underlying functionality, the post-change view id may still be semantically similar to the pre-change id. For example, `btnConfirm` may have been modified into `btnOK`. Lexical similarity cannot detect such relationships, so we introduce word embedding [6] to measure semantic distance between two view ids.

Since we want to exploit the semantic continuity in the functionality of the app (reflected in the view ids), we use an embedding model that is pre-trained for natural language corpus in English. However, view ids are often not single words: they are rather composite words made up of multiple tokens (such as `submitButton`). To cater for the composite word ids, we tokenise each identifier and apply the word embedding to the constituent tokens. More formally, given two view ids, i_1 and i_2, and a pre-trained word2vec embedding model, $w2v$, the semantic similarity $sem(i_1, i_2)$ is computed as follows, where cos_sim denotes cosine similarity:

$$\tau_1 = \{t : \text{ token } t \text{ is part of } i_1\}, \tau_2 = \{t : \text{ token } t \text{ is part of } i_2\}$$

$$sem(i_1, i_2) = \max_{(t_i, t_j) \in \tau_1 \times \tau_2} cos\_sim(w2v(t_i), w2v(t_j))$$

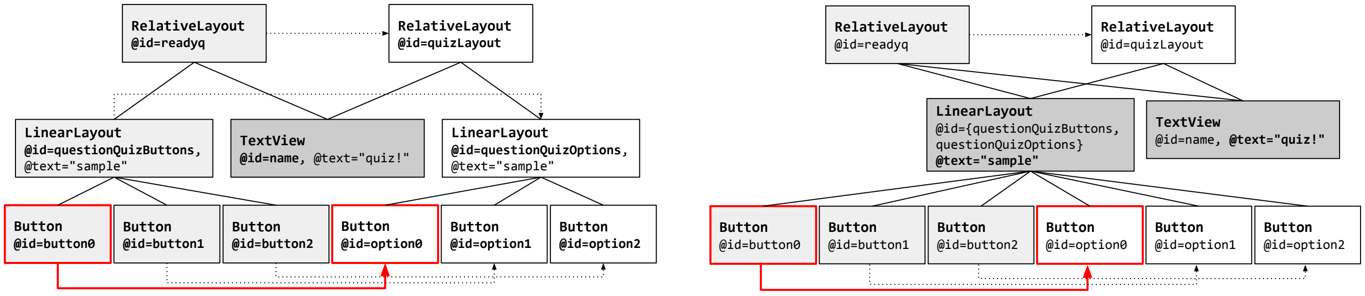
We hypothesise that, for a superficial view id change from i to i' that does not involve any significant change of the underlying app functionality, i and i' are likely to be semantically similar to each other, reflecting the continuity in app functionality. Given such similarity, it is also likely that tokens t and t' from i and i' respectively have high cosine similarity when embedded in vector forms. Note that when the changes made to the given pre-change view id is simply an addition of prefix or suffix, semantic similarity will produce the highest similarity, because both pre- and post-change view ids will include identical tokens.

D. Layout Similarity-based Input Features

Given the degree of freedom in natural language, we cannot guarantee that the use of lexical and semantic similarity is sufficient to capture all relationships between pre- and post-change view ids. For example, suppose the post-change view id entirely consists of an acronym that is not contained in the word embedding dictionary, or even non-English words: neither lexical nor semantic similarity computed via word embedding may cope well against such changes.

To assist identifier similarity features in such a situation, we introduce the structural similarity between GUI elements containing the pre- and post-change view ids in layouts. Layouts in an Android application define the structure for a user interface using a tree hierarchy of multiple GUI elements. Each GUI element corresponds to a node in the layout tree; in turn, each node contains multiple view properties such as `text`, `style`, `onClick`, `src` as well as `id`, as well as the type of the view, such as `Button` or `TextView`.

Analogous to our assumption about the continuity of AUT functionalities, we hypothesise that, despite cosmetic changes, pre- and post-change views will be placed in the same, or similar, locations within the hierarchy of the GUI. Consequently, we posit that using the similarity between surrounding layout context of GUI elements would help predict evolutionary



(a) `id` property based layout graph

(b) `text` property based layout graph

Fig. 1: Example layout graphs constructed for a pre-change id `button0` with different target property configurations. Lightgrey boxes indicate pre-change version layout, white boxes indicate post-change version layout, and darkgrey boxes are merged nodes of pre- and post-change elements. Simple lines indicate edges in the graph, and the arrows denote evolution links from pre- to post-change elements.

link from a pre-change GUI element to the post-change GUI element more precisely than using only identifiers.

1) Layout Graph Construction: To measure the structural similarity between the target GUI element referenced by a pre-change id and the GUI elements in the post-change version, we merge the layout trees in pre- and post-change versions of the AUT by identifying unchanged (and therefore overlapping) views, and measure the similarities between the pre-change node and the post-change candidate nodes. We use node2vec to measure the similarities between nodes using its node embeddings [**?**]. node2vec learns embeddings of nodes by maximising the likelihood of preserving network neighbourhood. As a result, node representation vectors are close to each other when the corresponding nodes share similar neighbour nodes (Network Homophily), or when they are located in similar substructures in the graph (Structural Equivalence). See Section **??** to see the details of how node2vec is used to measure the similarities.

Algorithm 1 explains how VIFER constructs a unified graph structure. The algorithm takes the pre-change and post-change version app source P and P', (specifically, files in the app containing the defined layouts), pre-change view identifier `preChangeId` that caused the failure of the test case to be repaired, and target properties `targetProps` to decide the identity of nodes in graph, i.e., GUI elements with the same values at all target properties are regarded as the same (see Section II-D1 for details of how we select target properties). For each statically defined layout file in app source code, the XML structure is imported as a tree (Line 5): we merge the nodes sharing selected `targetProps` to represent each node solely based on the selected GUI properties (Line 6). The trees with *generalised* nodes are composed iteratively (Line 13). Composing the two graphs results in merging the nodes present in both graphs, and with the same values for `targetProps`. For the pre-change version, only the layout trees containing `preChangeId` are considered for the final graph.

A similar process is applied for every post-change version layout to merge nodes and compose trees (Line 14-16). Note that the resulting graph G can include disconnected components. This is because there are cases where the layout that contains the pre-change view does not have any overlapping node with the layout that contains the candidate post-change view with respect to a specific target property configuration. This results in no edges between these two layouts. Disconnected components are prone to occur when we set more target properties (e.g., *all* configuration in Section II-D1) to distinguish nodes. We extract connected graphs containing one pre-change node (i.e., a node that has id property of `preChangeId`) and all of the nodes that are reachable from the pre-change node and edges between them (Line 18).

2) Target Property Selection: In VIFER, layout graphs are constructed multiple times based on different subsets of GUI properties listed below. Among the diverse properties used in Android GUIs, we choose five properties: `id` (the view identifier), `text` (the text displayed on the view, such as the labels on buttons), `style` (the string definition of its visual style, `onClick` (the name of the onClick event handler), and `src` (the name of the graphical resource used by the view). We additionally include the type used to define the GUI elements (e.g., `Button`, `TextView`) and treat it as a property.

We consider each of these properties as the node representation, as well as the following four combinations of properties:

- (*all*) Use all selected properties: {`id`, `text`, `src`, `onClick`, `style`, `type`}
- (*all-except-id*) Use all selected properties except for id: {`text`, `src`, `onClick`, `style`, `type`}
- (*visual*) Use visual-related properties: {`src`, `style`, `type`}
- (*content*) Use content-related properties: {`onClick`, `text`}

For example, under the *all* properties configuration, a node is considered to be identical to another only when all six properties are equal. However, under the *visual* properties configuration, any two nodes that share the same `src`, `style`, and `type` property values are considered to be identical. Using a range of target properties allows us to consider the structural contexts in given GUI layout trees from diverse perspectives. Figure **??** presents an example substructure of the layout graphs built on two different properties for representing nodes: `id`, and `text` respectively. Consider the element with the id

Algorithm 1: Layout graph construction to learn node embedding.

```
CreateGraph ($P, P', preChangeId, targetProps$)
    Inputs : Pre-change version app $P$; Post-change version app
             $P'$; Pre-change view identifier from the failed test case
             $preChangeId$; Target properties set $targetProps$;
    Output: Set of graphs of GUI elements both from $P$ and $P'$.
    $G \leftarrow graph()$;
    $G_{set} \leftarrow set()$;
    $PreNodes \leftarrow set()$;
    foreach $T \in getLayoutTrees(P)$ do
        $T = mergeNodes(T, targetProps)$;
        $hasPreChangeNode \leftarrow False$;
        foreach $N \in T.nodes$ do
            if $N.hasId(preChangeId)$ then
                $PreNode.add(N)$;
                $hasPreChangeNode \leftarrow True$;
        if $hasPreChangeNode$ then
            $G \leftarrow compose(G, T)$;
    foreach $T \in getLayoutTrees(P')$ do
        $T = mergeNodes(T, targetProps)$;
        $G \leftarrow compose(G, T)$;
    foreach $N \in PreNodes$ do
        $G_{set}.add(connectedSubgraph(G, N))$;
    return $G_{set}$;
mergeNodes ($T, targetProps$)
    foreach $N_i, N_j \in T.nodes$ do
        if $N_i.get(targetProps) \equiv N_j.get(targetProps)$ then
            $N_i.parents \leftarrow N_i.parents \cup N_j.parents$;
            $N_i.children \leftarrow N_i.children \cup N_j.children$;
            $N_j.parents \leftarrow set()$;
            $N_j.children \leftarrow set()$;
```

`button0` in the pre-change version, which is changed to the element with id `option0` in the post-change version. As shown in Figure **??**, if we construct a graph using Algorithm 1 and merge nodes by the `id` property, the actual pre- and post-change node containing the changed view ids do not show considerable similarity. This is because the `id` of the parent element of the two nodes has changed as well, thereby also changing the neighbouring contexts of these nodes. In contrast, for the layout graph based on `text` property in Figure **??**, pre- and post-change nodes share the same parent because the parent maintains the same `text` attribute across the change.

3) Structural Similarity Feature using Node Embeddings: For each graph including a pre-change node, we train a node2vec model to generate node embeddings, and measure cosine similarity between the embeddings of the pre-change node and any other nodes to which any `id` property is assigned.[2] Given a subgraph $g \in G_{set}$ returned by Algorithm 1 as well as a pair of view ids (i.e., pre-change id and a candidate post-change id), a and b, we first compute the cosine similarity between node n_a and n_b in g, which are nodes whose id is a and b, respectively. Since there may be multiple views with the same id,[3] we consider sets of nodes with the id a and b:

$$N_{a,g} = \{n | n \in g \wedge id(n) = a\}, N_{b,g} = \{n | n \in g \wedge id(n) = b\}$$

The structural similarity between ($N_{a,g}$ and $N_{b,g}$), based on property p on g is defined as:

$$\sigma_p(N_{a,g}, N_{b,g}) = \max_{(n_a,n_b) \in N_a \times N_b} cos\_sim(n2v(n_a), n2v(n_b))$$

Finally, we compute the structural similarity based on property p between id a and b as follows:

$$str\_sim(a, b, p) = \max_{g \in G_{set}} \sigma_p(N_{a,g}, N_{b,g})$$

E. Text Property Similarity-based Input Features

One property that can reflect the functionality of the view is the resource id handle that is used to load the text labels from the resource pool to the view. To support internationalisation, a `text` property value is typically assigned to a GUI element from the pool of predefined string resources. As such, the resource id used as the alias to the actual string value may reflect the functionality of the view that receives the text label. Consequently, based on the same continuity hypothesis, we expect that the post-change resource id to be similar to the pre-change id.

If the `text` property values (either resource ids or string literals) between pre- and post-change views are identical, structural similarity based on `text` property can capture the relevance, as pre- and post-change views will be merged in the graph, resulting in a similarity of 1.0. However, there may be changes that affect the text property itself. For example, a view id change from `@+id/buttonLinkDropbox` to `@+id/buttonSetupSync` can accompany the change of text property from the resource id of `@string/configure_dropbox` to a new resource id of `@string/configure_sync`. To handle such changes, we introduce a similarity feature for the `text` property: essentially, this is the semantic similarity between `text` property values.

F. Ranking using Cascaded Classifiers

Our classification model uses a total of 13 features in three major groups described in the previous section: lexical and semantic similarity on view ids (2), structural context similarity on layouts (6 individual similarity metrics, plus 4 target property configurations), and semantic similarity on the text property (1). While the view id and layout-based similarities can be measured between any pair of pre- and post-change views, the text property-based similarity feature could be missing when the pre-change view does not have the text property value. Consequently, our technique contains two separate classifiers, one that includes the text property-based similarity as an input feature, and the other that does not. Given a classification algorithm A, we use the following classification models A_2 and A_3 depending on whether the text property-based similarity feature is available or not:

[2] Since we are aiming to resolve a VIF, by definition we are looking for a post-change view that has an id.

[3] This is possible when these views exist in different Android layout files.

- A_2: the classifier A_2 is an instance of algorithm A that uses view id-based similarity and layout-based similarities as input features.
- A_3: the classifier A_3 is an instance of algorithm A that uses all three types of input features: view id-based similarity, layout-based similarity, and text property-based similarity.

Suppose we choose Logistic Regression (LR) as our classification algorithm. Given a pair of pre- and post- change view ids, the cascaded classifiers will first check whether any GUI element with the pre-change id has text attributes: if so, LR_3 will be to classify whether the pair of ids points to the same GUI widget or not and produce prediction scores for further ranking. If not, it will apply LR_2.

G. Test Case Repair

We evaluate our GUI test case repair technique by simulating VIFs in real-world Android applications and subsequently applying our view id repair based on rankings produced from cascaded classifiers. We first collect view id changes (i.e., pairs of pre- and post- change ids) from the version history of each app. Using the changes, VIFs can be simulated by replacing the post-change view id in the test case to the corresponding pre-change view id. The list of Android applications used for constructing repair targets are presented in Section III-B. For the sake of simplicity, we make the single VIF assumption, i.e., that there only exists a single VIF in a test case without any other test failures. Hence, it becomes possible to decide whether the simulated VIF has been successfully repaired simply by checking whether the test case broken by VIF passes after replacing the outdated pre-change view id with the matched post-change view id. Below are the repair steps:

1) **Localisation of VIF:** Using the failure message either caused by a compiler error or a runtime `ViewMatchingException`, we localise the exact source code location of the outdated view identifier.
2) **Identification of GUI layout context and properties:** We try to extract GUI layout files from both for pre- and post- change versions. Currently, we use static layout files contained in the resource directory of app source code. To combine the pre- and post-change layouts, we apply the graph construction process described in Section II-D. Afterwards, we build the candidate view id pool in the post-change version (see Section III-B for more details about how we construct the candidate view id pool). We can also identify whether a `text` property exists for the pre-change view id. Depending on the existence of the `text` property, we determine which model to use as described in III-C.
3) **Ranking candidate identifiers by model:** According to the possible number of features identified from the previous step, the features are dynamically measured and fed into the cascaded classification model.
4) **Iterative repair:** Once the list of the ranked candidate view ids is generated, our repair technique iteratively replaces the outdated view identifier with the candidate id in order, and terminates when the test case passes. We heuristically consider that the repair attempt was successful if the target test case passes.

The performance of our repair technique largely depends on the accuracy of our classification model, as the essence of the repair is the update of the view id. We still opt to evaluate the repair process separately, to investigate how effective the supporting steps (such as localisation and iterative repair) are.

III. Experimental Setup

This section describes the details of our experimental setup for the empirical evaluation.

A. Research Questions

We aim to answer the following research questions to evaluate our proposed approach.

RQ1. Similarity Effectiveness: How effective are the individual similarity metrics described in Section II-C? We answer RQ1 by sorting all post-change candidate view ids according to each individual similarity feature and checking the rank of the correct post-change candidate. We use $acc@n$ as the evaluation metric, which is the number of pre-change view ids for which our technique can rank the correct post-change counterpart within the rank of n. Ties are broken using max tie-breaking.

RQ2. Model Effectiveness: How effective is the classification model trained on multiple features when compared to individual similarity features? We study classification models with different input feature configurations: only identifier-based similarity, identifier and layout-based similarity, identifier, layout, and text property-based similarity. We answer RQ2 by verifying whether that adding more similarity features from diverse sources linked to the view ids improves post-change view identification. As before, we rank the post-change candidate ids using the prediction scores obtained by classification models, and compare the models using $acc@n$ evaluation metric.

RQ3. Repair Performance: How effective is VIFER for repairing View Identification Failure in realistic GUI testing settings? We answer RQ3 by applying the test case repair described in Section II-F to the failing GUI test cases due to VIF. We report the rank of the correct post-change id that makes the broken test cases pass among all identified candidate post-change ids.

B. Subjects

Table I contains ten open source Android apps used in our study. All of these apps use the Espresso test automation framework, and have been studied by Coppola et al. [4] for Android test fragility. From the repositories of these ten apps, we extract 376 commits that modify only view ids and not the app code, as these commits are likely to result in VIF. The 376 commits provide a total of 512 pairs of pre- and

TABLE I: List of subject Android apps

Name	# of Commits	# of ID Changes	w/ Text Prop.
Pr0	20	20	7
akvo-caddisfly	51	144	78
alltrack	1	2	2
andFHEM	22	47	27
android	156	148	70
android-money-manager-ex	33	36	13
android_Skeleton	11	23	3
cominghome	2	2	1
poly-picker	2	15	4
ts-android	78	75	47
Total	376	512	252

post-change view ids, (i, i'), as well as negative pairs of the other post-change view ids that are also contained in the post-change version. These 512 id change cases form the basis of our empirical study for RQ1 and RQ2.

While all such pairs can provide the textual (i.e., lexical and semantic) similarity between two view ids, not all pairs yield connected GUI elements with text property. Table I also shows how many pairs provide the additional text property-based similarity features described in Section II-C.

To evaluate our classification approach for RQ2, we also need to define the pool of post-change candidate view ids. we take all post-change view ids that appear in the GUI layout resource of the AUT and consider all of them as the candidate post-change view id pool.

We apply ten-fold cross validation for evaluation of models. The reported model performance is aggregated across the results from each fold: the result for each pair of (i, i') is taken from the run in which the fold that contained the given pair was used for validation.

TABLE II: List of Android apps used for simulating repair

Name	# of TC	Repository URL
blabbertabber	3	https://github.com/blabbertabber/blabbertabber
KISS	5	https://github.com/Neamar/KISS
JustBe-Android	6	https://github.com/justbeneu/JustBe-Android
gini-vision-lib-android	1	https://github.com/gini/gini-vision-lib-android
NoiseCapture	1	https://github.com/Ifsttar/NoiseCapture
Total	16	

To answer RQ3, we emulate realistic VIF scenarios using view id changes we find in open source Android apps. We use a separate set of five open source Android apps, described in Table **??**: these apps are also studied by Coppola et al. [4]. Due to various version compatibility issues, we emulate test cases broken by VIF, instead of mining their timeline to find exact VIFs. For this, we first identify a pair of pre- and post-change view ids, (i, i'), that actually took place during the lifetime of AUT. Subsequently we identify executable GUI test cases, written by the original developers, that refer to i'. Finally, we seed a VIF by replacing i' with i. The cascaded model used in RQ3 is trained on the full dataset in Table I and applied to the separate repair target applications in Table **??**.

C. Metrics, Models & Environments

We measure the lexical similarity (i.e., Levenshtein distance), using the `nltk` library [8]. To measure semantic similarity, we use `spiral` [**?**] to tokenise view ids and the pre-trained word2vec embedding model[4] to measure the semantic distances between tokens. As explained in Section II-C, the similarity between two view ids is the maximum similarity between subtokens from each view id.

As a classification model, we use Logistic Regression with `lbfgs` solver and L2 regularisation implemented in `scikit-learn` version 0.23.2 [9]. The empirical evaluation was conducted on machines equipped with Intel Core i7 and 32GB of RAM. We use the word2vec embedding model in the `gensim` library [11], and an open source Python implementation[5] of node2vec.

IV. RESULTS

This section presents the results from our empirical evaluation and answers the research questions.

A. RQ1. Similarity Effectiveness

Table II shows the $acc@n$ with $n \in \{1, 3, 5, 10\}$ by ranking the post-change ids solely based on individual similarity metrics. Note that the total number of pre-change ids that yield these similarity values are different: we can measure id- and layout-based similarity for all 512 studied pre-change view ids, but text-based similarity for only 252 view ids due to the unavailable text properties in some elements. For fairer comparison between different similarity features, we also report all $acc@n$ results as percentages against the total number of relevant view ids (which is shown in the Total row).

We compare all individual features first. The results in Table II shows the $acc@n$ values of all 13 individual features. The results show that id-based semantic similarity performs the best among the individual features, according to $acc@n$ metrics where $n \in \{3, 5, 10\}$. For $acc@1$, structural similarity feature based on the *all-except-id* configuration (see Section II-D1) performs the best, placing 201 out of 512 VIF pairs at the top.

However, a closer analysis also reveals that these similarity features are complementary. Figure 2 contains Venn diagrams of $acc@1$ and $acc@10$ results produced by these four features (note that structural similarity features are represented by the best performing *all-except-id* configuration to avoid visual clutter in the diagram). While there is a large intersection, each individual similarity feature also makes unique view identifications, except for the fact that the text-based similarity does not uniquely identify any view id changes in $acc@1$.

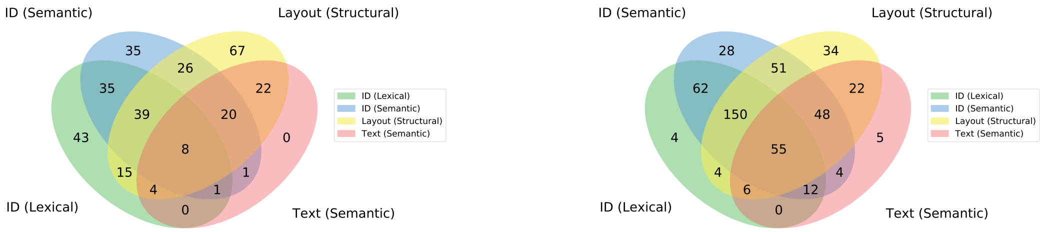
The complementary nature of these features can also be observed when we zoom in to compare different target property configurations for the structural similarity features. In Table II, among the six individual view properties (`id`, `type`, `text`, `style`, `src`, and `onClick`), structural similarity based on `text`

[4] https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM

[5] https://github.com/eliorc/node2vec

TABLE III: Accuracy@n for individual similarity features and their percentage against total number of pre-change ids that yield the corresponding similarity feature values.

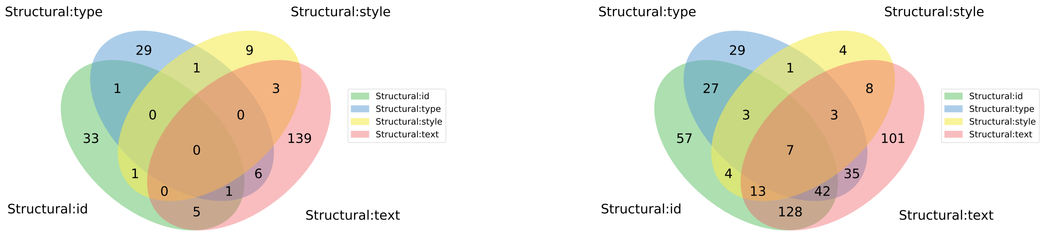
Similarity		*acc*@1		*acc*@3		*acc*@5		*acc*@10		Total	
`Structural`	`id`	41	(8.0%)	126	(24.6%)	183	(35.7%)	281	(54.9%)	512	(100%)
	`type`	38	(7.4%)	78	(15.2%)	100	(19.5%)	147	(28.7%)	512	(100%)
	`text`	154	(30.1%)	244	(47.7%)	281	(54.9%)	337	(65.8%)	512	(100%)
	`style`	22	(4.3%)	61	(11.9%)	91	(17.8%)	152	(29.7%)	512	(100%)
	`src`	42	(8.2%)	76	(14.8%)	103	(20.1%)	138	(27.0%)	512	(100%)
	`onClick`	14	(2.7%)	20	(3.9%)	29	(5.7%)	43	(8.4%)	512	(100%)
	`all`	62	(12.1%)	147	(28.7%)	202	(39.5%)	267	(52.1%)	512	(100%)
	`all-except-id`	**201**	**(39.3%)**	284	(55.5%)	320	(62.5%)	370	(72.3%)	512	(100%)
	`visual`	73	(14.3%)	134	(26.2%)	167	(32.6%)	240	(46.9%)	512	(100%)
	`content`	161	(31.4%)	241	(47.1%)	270	(52.7%)	324	(63.3%)	512	(100%)
`ID`	`lexical`	145	(28.3%)	221	(43.2%)	247	(48.2%)	293	(57.2%)	512	(100%)
	`semantic`	165	(32.2%)	**317**	**(61.9%)**	**359**	**(70.1%)**	**410**	**(80.1%)**	512	(100%)
`Text`	-	56	(22.2%)	105	(41.7%)	123	(48.8%)	152	(60.3%)	252	(100%)



(a) *acc*@1

(b) *acc*@10

Fig. 2: Venn diagram of *acc*@1 and *acc*@10 results produced by individual similarity features.



(a) *acc*@1

(b) *acc*@10

Fig. 3: Venn diagram of *acc*@1 and *acc*@10 results produced using four different types of layout-based structural similarity.

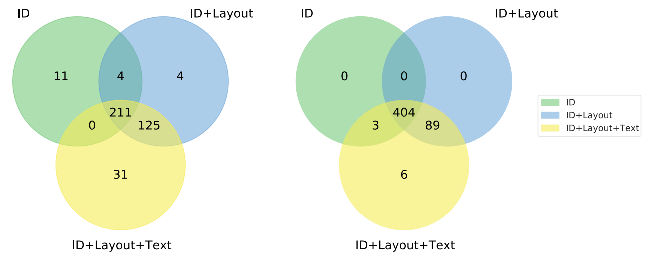
property achieves the best ranking performance. However, Figure **??** shows that different configurations can make their own contributions to the final results.

TABLE IV: Accuracy@n from classification models using Logistic Regression, trained on different sets of features.

Accuracy	`ID`	`ID+Layout`	`ID+Layout+Text`
acc@1	226 (44.1%)	344 (67.2%)	**367 (71.7%)**
acc@3	339 (66.2%)	437 (85.4%)	**453 (88.5%)**
acc@5	370 (72.3%)	469 (91.6%)	**480 (93.8%)**
acc@10	407 (79.5%)	493 (96.3%)	**502 (98%)**
Total	512 (100%)	512 (100%)	512 (100%)

B. RQ2. Model Effectiveness

Table III contains the results obtained from classification models based on different sets of features: best results are typeset in **bold**. We compare a model that uses id-based lexical and semantic similarities (`ID`), a model that uses ten structural similarity features (six individual similarities and four target property configurations) in addition to ID (`ID+Layout`), and a model that uses text-based semantic similarity in addition to the others (`ID+Layout+Text`). Compared to the ID model, the `ID+Layout` model shows significantly improved accuracy, showing that the inclusion of structural similarity helps identifying the correct post-change view ids. Additional inclusion of text-based similarity feature further improves the ranking



(a) $acc@1$ (b) $acc@10$

Fig. 4: Venn diagram of $acc@1$ and $acc@10$ results produced by cascaded classification models using different scope of similarity features.

performance, but the margin of improvement is not as large as that of the ID+Layout model over the ID model. Note that the ID+Layout+Text model is a cascade model, meaning that the text based similarity feature is only used when available.

Figure 3 shows a detailed break-down of $acc@1$ and $acc@10$ values achieved by different models. We observe that ID+Layout+Text model subsumes the other two models for $acc@10$. However, with $acc@1$, all three models contribute more evenly to the final results.

TABLE V: Result of VIF repair in real-world GUI test cases.

Project Name	Pre-change ID	Post-change ID	Rank
blabbertabber	dummy_stop_button dummy_finish_button acknowledgementsText	button_pause_caption button_finish aboutText	3 1 1
KISS	main_kissbar favoritesBar launcher searchTextField favoritesKissBar	mainKissbar externalFavoriteBar launcherButton searchEditText embeddedFavoritesBar	1 2 1 6 1
JustBe-Android	submitButton tenText dropdown_spinner editRepeatPassword editLasttName zeroText	nextButton maxText sleep_spinner editConfirmPassword editUsername minText	2 2 1 6 9 3
gini-vision-lib-android	gv_viewpager	gv_onboarding_viewpager	1
NoiseCapture	linearLayout2	graph_components_layout	2

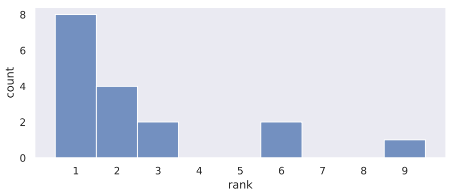


Fig. 5: Distribution of repair ranks. (16 test cases)

C. RQ3. Repair Performance

We apply our test case repair technique, described in Section II-F, on emulated VIFs following the process described in Section III-B. Table **??** presents repair result for each VIF, with the pre-change id seeded to simulate VIF and the repaired post-change id that was found by VIFER. Among 16 failing GUI test cases, seven were fixed within only a single attempt (i.e., the correct post-change id is ranked at the top). On average, a successful repair requires 2.62 repair attempts: all of the 16 test cases could be repaired within ten id replacement attempts. The distribution of all ranks are shown in Figure **??**.

Listing **??** shows the body of a repaired test case from a single attempt and applied patch. The softmax scores produced by the classifier for this repair are shown in Figure **??**: the correct post-change view id is chosen with high confidence.

```
@Test
public void testViewDisplay() {
-   onView(withId(R.id.dropdown_spinner))
+   onView(withId(R.id.sleep_spinner))
        .check(ViewAssertions.matches(isDisplayed()));
}
```

Listing 1: Example of a test case repaired the first attempt.

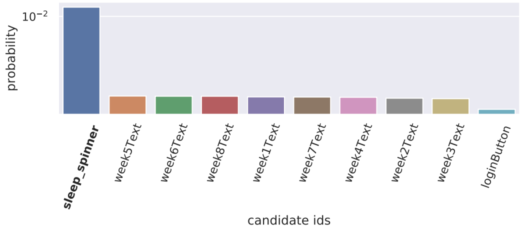


Fig. 6: Softmax probabilities of the top ten candidate ids, obtained by classification model to repair the test case in Listing **??** (probability is adjusted to a log scale).

Another test case, shown in Listing **??**, is repaired in the sixth attempt. Figure **??** shows that the softmax score for the correct post-change id, searchEditText, is lower than those of the other five candidate ids. A closer investigation reveals that, due to the disconnected subgraphs used in structural similarity computation, some of the similarity features are zeroes for this particular view id, which affected the final classification results significantly.

```
@Test
public void testCanTypeTextIntoSearchBox() {
-   onView(withId(R.id.searchTextField))
+   onView(withId(R.id.searchEditText))
        .perform(typeText("Test"))
        .check(matches(withText("Test")));
}
```

Listing 2: Example of a test case repaired in the sixth attempt.

V. Threats to Validity

Threats to internal validity concern factors that can affect the observed effects, such as the correctness of the various similarity metric values computed between GUI elements and their properties. We rely on widely used public implementations (scikit-learn [9], nltk [8], node2vec [**?**]) and pre-trained word embeddings (gensim [11]) to mitigate these concerns. Threats to external validity concern factors that may prevent

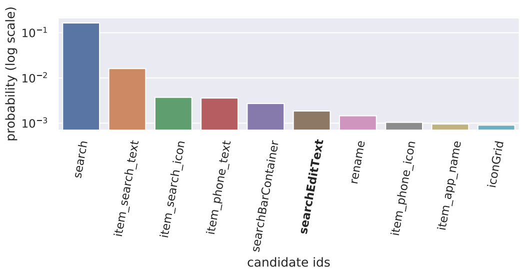


Fig. 7: Softmax probabilities of the top ten candidate ids, obtained by classification model to repair the test case in Listing **??** (probability is adjusted to a log scale).

wider generalisation of our results. Our study considers a subset of open source Android apps previously studied by Coppola et al. [3] as it contains a large available catalogue of VIFs observed in the wild. We also limit our scope to open source apps due to licence concerns, as our current technique requires internal information from apps, such as the GUI layout tree in the form of XML: barring complete reverse engineering, different approaches to extract GUI layout may produce different features from the ones we studied. To ensure generalisability as much as possible, we try to maintain our repair scenario as realistic as possible: we reuse existing GUI test cases with minimum purification, and only use VIF inducing view id changes that have actually happened in the development history. However, our results are still specific to the studied apps. For example, the quality of original GUI test cases can affect our findings: if the original test cases are weak, our repair may simply be plausible and not correct. Finally, threats to construct validity concern whether what we measure actually reflects what we want to observe. All our observations are based on simple count based metrics that are easy to interpret, as well as publicly reviewed ground truth (Coppola et al. [3]).

VI. Related Work

VIF is part of GUI test case fragility [3]: by fragility, we mean that the test case can be easily broken by commits to the program under test that are not bug inducing. It can be considered as a specific form of test flakiness [19], [20], which refers to test cases whose outcome changes due to reasons unrelated to the correctness of the program under test (e.g., unintended nondeterminism, or randomness in environmental factors). However, fragility tends only to break test cases, whereas flakiness in general means the test result oscillates between pass and fail randomly. While there are studies that investigates the relationship between test smells and flakiness [21], a direct repair of test flakiness remains challenging.

While automated repair of programs under test has been widely studied [12], [13], [14], [15], [16], automated repair of test cases remains relatively unexplored. A closely related topic is that of test augmentation, which aims to augment an existing test suite so that it becomes adequate against an evolved program [17], [18]. However, augmentation typically concerns making a test suite more adequate by adding new test cases, whereas our approach aims at repairing individual test cases by resolving VIFs.

WATER [7] uses structural information to repair GUI test script of web applications. When a VIF is encountered, it tries replacing the *lost* GUI element with any web elements that has the same value for at least one key property. However, repair may be infeasible when there is no GUI element with the exactly the same set of property values. To avoid such brittleness, we use structural contexts, which can consider a wider range of candidates. Recently suggested techniques such as METER [**?**] and GUIDER [**?**] aim to repair obsolete Android GUI test scripts, similarly to VIFER. To match the GUI elements, METER focuses on visual similarity using computer vision techniques; GUIDER adds structural information to METER. However, GUIDER only compares the information contained in a single GUI element by matching its property values, whereas VIFER includes both element-wise similarity and the neighbouring contexts to match view ids.

VII. Conclusion

We present a machine learning based test case repair technique for View Identification Failures (VIFs) to address Android GUI test case fragility issue in an industry relevant setting. Our test scenario concerns GUI level smoke test, in which third-party apps are executed using GUI test scripts written by smartphone vendors to ensure compatibility of vendor specific customisations. In such a scenario, since the test scripts are written in isolation from app development, test cases are fragile against VIFs. Based on the assumption that changes made to view ids are not deeply coupled to changes in the functionality of the app, we hypothesise that pre- and post-change view ids will be similar to each other, and evaluate different similarity measures to match pre- and post-change view ids. We formulate the problem as classification of pre- and post-change view id pairs, and evaluate a set of similarity features that can be measured from Android GUI views. Empirical evaluation of our classification model shows that it can precisely match about 71.7% of view id changes collected from real world open source Android apps at the top of the rank, and can locate the correct post-change view identifier within top 10 candidates for 98% of the cases. This is a significant improvement over the currently deployed technique that is based on lexical similarity only, which matches only 28.3% of the view id changes at the top.

Acknowledgement

This work is supported by Samsung Research, Samsung Electronics Co., Ltd. Juyeon Yoon, Seungjoon Chung, Jinhan Kim, and Shin Yoo are partly supported by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629). Shin Hong is partly supported by National Research Foundation of Korea (NRF) Grant (NRF-2021R1A5A1021944).

References

[1] IDC, "Smartphone market share https://www.idc.com/promo/smartphone-market-share/os (Last checked: 22 November 2020)," September 2020.

[2] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 226–237. [Online]. Available: https://doi.org/10.1145/2970276.2970312

[3] R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile ui test fragility: An exploratory assessment study on android," in *Proceedings of the 2nd International Workshop on User Interface Test Automation*, ser. INTUITEST 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 11–20. [Online]. Available: https://doi.org/10.1145/2945404.2945406

[4] R. Coppola, M. Morisio, M. Torchiano, and L. Ardito, "Scripted GUI testing of android open-source apps: evolution of test code and fragility causes," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3205–3248, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09722-9

[5] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[7] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, 2011, pp. 24–29.

[8] E. L. Bird, Steven and E. Klein, *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[9] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[10] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. Madison, WI, USA: Omnipress, 2010, pp. 807–814.

[11] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[12] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 947–954.

[13] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 772–781.

[14] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 691–701.

[15] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[16] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180233

[17] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sept 2008, pp. 218–227.

[18] Z. Xu, "Directed test suite augmentation," in *Proceedings of the 33rd International Conference on Software Engineering*, May 2011, pp. 1110–1113.

[19] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 534–538.

[20] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[21] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 1–12.