Differencing Graphical User Interfaces

Mark Grechanik, Chi Wu Mao, Ankush Baisal
University of Illinois at Chicago
Chicago, IL 60607
{drmark,cmao4,abansa6}@uic.edu

B.M. Mainul Hossain
Institute of Information Technology
University of Dhaka, Bangladesh
mainul@iit.du.ac.bd

David S. Rosenblum
School of Computing
National University of Singapore
david@comp.nus.edu.sg

***Abstract*—Graphical User Interface *(GUI)-based APplications (GAPs)* are ubiquitous and provide a wealth of sophisticated services. Nontrivial GAPs evolve through many versions, and understanding how GUIs of different versions of GAPs differ is crucial for various software quality tasks such as testing, cross-platform UI comparison and project effort estimation. Yet despite the criticality of automating GUI differencing, it is a manual, tedious, and laborious task.**

We offer a novel approach for differencing GUIs that combines tree edit distance measure algorithms with accessibility technologies for obtaining GUI models in a non-intrusive, platform and language-independent way, and it does not require the source code of GAPs. We developed a tool called *GUI DifferEntiator (GUIDE)* that allows users to difference GUIs of running GAPs. To evaluate GUIDE, we created an experimental platform that generates random GUIs with controlled differentials among them that serve as oracles. GUIDE enables researchers to plug-and-play various GUI differencing algorithms and to automatically run experiments. We evaluated GUIDE on 5,000 pairs of generated complex GUIs and three open-source GAPs and the results of our evaluation suggest that GUIDE can find differences between GUIs with a high degree of automation and precision.

***Index Terms*—differencing, GUI, tree-edit, algorithms, accessibility, testing, similarity, generator, comparison**

I. Introduction

Applications with *Graphical User Interface (GUIs)* are ubiquitous and provide a wealth of sophisticated services. Nontrivial GAPs contain hundreds of GUI screens and thousands of GUI objects. GAPs evolve through many versions and they are run on different platforms, and understanding differences between GUIs is crucial for various software quality tasks such as testing, cross-platform UI matching, GAP rejuvenation and project effort estimations.

Consider test automation, which plays a key role in reducing high cost of testing GAPs [1][2][3]. To automate testing process, engineers write programs using scripting languages (e.g., JavaScript and VBScript), and these programs (*test scripts*) mimic users by performing actions on GUI objects of these GAPs using some underlying testing frameworks [4], [5]. An extra effort put in writing test scripts is paid off when these scripts are run repeatedly to determine if GAPs behave as desired. However, an impediment to test automation is that releasing new versions of GAPs with modified GUIs breaks their corresponding test scripts [6], [7], [8]. As many as 74% of the test cases become unusable during GUI regression testing [9]. To reuse these scripts, test engineers should fix them, and this process involves comparing GUIs to find changed GUI objects, and subsequently operations in test scripts that reference these objects.

Knowing how many fixes are required for test scripts is part of project effort estimation during various software quality tasks, and project effort estimation tasks are based on differencing GUIs, for example, during prototyping. In the spiral model of software development [10], a prototype is built and shown to different stakeholders in order to receive an early feedback [11]. This feedback often leads to changes in the prototype and the original requirements as stakeholders refine their vision. In order to estimate development and testing effort, GUIs of the prototypes should be compared in order to determine how they differ. Unavailability of automated tools results in significant manual effort when comparing GUIs.

With the volume of mobile application development overtaking the server-side [12], GAPs are essential on most mobile devices that host different GUI frameworks. As a result, the GUI layouts often appear different on different mobile devices, due to platform differences (operating systems, browsers, etc) and device characteristics such as screen dimensions. Determining such differences manually is a laborious, imprecise and expensive effort. Thus, automating GUI differencing is an important problem that affects many aspects of software evolution and maintenance.

Differencing GUIs is a difficult and generally unsolved problem. Well-established GUI frameworks (e.g., Motif, MacOS, Windows) contain hundreds of different types of GUI objects. Some of them are basic (e.g., buttons or lists) and the others are complex (e.g., tables or container objects that contain other GUI objects). A simple and brute-force algorithm for differencing GUIs is to convert them into images and perform pixel by pixel comparison. Unfortunately, this approach is ineffective in general, since even a slightest modification to the GUI (e.g., changing the color of a pixel) results in mismatches, leading to a large number of warnings about mismatched GUI objects and false positives. To the best of our knowledge, there is no solution to GUI differencing that is general to be applied to all major GUI frameworks, non-intrusive (e.g., does not require programmers to insert annotations in the code that implements GAPs), platform and language-independent, and that does not require the source code of the GAP.

We offer a novel approach for differencing GUIs. Our key insights is to extract models of GUIs using the generic interfaces of the *accessibility technologies* that are an integral part of GUI frameworks since they are mandated by laws of most countries. GUIs are modeled as ordered rooted trees with labeled vertices and we extract these models from running GAPs automatically using accessibility technologies [4]. A main contribution of our paper is in applying and evaluating tree edit distance algorithms to difference GUIs by computing the minimum cost solution of transforming one tree into the other by a sequence of elementary operations consisting of deleting and relabeling existing GUI objects, as well as inserting new objects. Our approach is non-intrusive, platform and language-independent, and it does not require the source code of GAPs. We believe that ours is the first automatic approach to difference GUIs in a systematic way.

We created an experimental platform called *GUI DifferEntiator (GUIDE)* for evaluating our approach. Using GUIDE toolkit, we generate random GUIs with controlled differentials among them that serve as oracles, and we enable researchers to plug-and-play various differencing algorithms and to automatically run experiments in parallel and produce results. We evaluated GUIDE on few open-source GAPs and 5,000 generated GUIs using our platform. The results of our evaluation suggest that our approach can find differences between GUIs with a high degree of automation and precision.

II. The Problem Statement

In this section, we give background on GUI frameworks, show the structure of GAPs, describe the results of our empirical study with GUI differencing in the context of software testing, and formulate the problem statement.

A. Background on GUI Frameworks

A GUI framework is a reusable and extensible set of GUI objects with well-defined interfaces that can be specialized to produce and to run custom GAPs [13]. A schematics of a model of GUI frameworks is shown in Figure 1. A purpose of this model is to show that there are four main components in GUI frameworks: GUI representation layer, GUI object library, GUI interfaces and the accessibility layer. GUI representation layer defines how GUI objects are represented programmatically as data structures in computer memory. For example, HTML pages are represented using a generic document object model in browsers whereas the programming documentation for Windows defines proprietary data structures for GUIs and the events that they receive and send. The visualization/graphics engine interprets these data structures and visualizes them using some predefined settings for styles and layouts [14], [15].

To allow users to interact with GUI objects, the underlying operating system or a virtual machine provides queues for receiving user inputs from peripheral devices (e.g., mouse, keyboard or a touch screen) and translates these inputs into event data structures that are passed to the corresponding GUI objects using its interfaces. GUI object libraries contain implementations of GUI objects and expose their interfaces; these libraries are extensible and many third-party vendors offer implementations of sophisticated GUI objects for different GUI frameworks. In general, GUI frameworks, which are developed by different vendors, expose diverse interfaces.

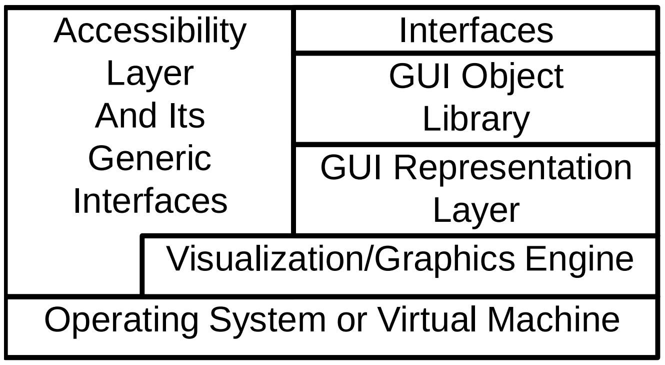
Accessibility Layer And Its Generic Interfaces
Interfaces
GUI Object Library
GUI Representation Layer
Visualization/Graphics Engine
Operating System or Virtual Machine

Fig. 1. A model of GUI Frameworks.

Accessibility technologies provide diverse aids to disabled computer users [16][17]. Specific aids include screen readers for the visually impaired, visual indicators or captions for users with hearing loss, and software to compensate for motion disabilities. Most computing platforms include accessibility technologies since electronic and information technology products and services are required to meet the *Electronic and Information Accessibility Standards* [16]. For example, *Microsoft Active Accessibility (MSAA)* technology is designed to improve the way accessibility aids work with applications running on Windows [18], [19], and *Android Accessibility* technology assists disabled users who run mobile applications[1]. As mandated by laws[2], accessibility technologies are incorporated into most computing platforms.

Accessibility technologies provide a wealth of generic sophisticated services required to retrieve attributes of GUI objects, set and retrieve their values. A key element of accessibility technologies is that they provide a generic set of *Application Programming Interface (API)* calls that different GUI libraries should implement as a common programming standard for applications that use the accessibility layer. For example, all Windows GUI objects should implement the `IAccessible` interface in order to be accessed and controlled using the MSAA API calls. Programmers may write code to access and control GUI objects of GAPs as if these objects were standard programming objects. Using accessibility technologies, programmers can also register callback functions for different events produced by GUI objects thereby obtaining timely information about states of the GUI objects of the GAPs. For example, if a GUI object receives an incorrect input and the GAP shows an error message dialog informing the user about the mistake, then a previously registered callback can intercept this event signaling that the message dialog is being

[1]http://developer.android.com/guide/topics/ui/accessibility/index.html
[2]http://www.ada.gov/2010ADAstandards_index.htm

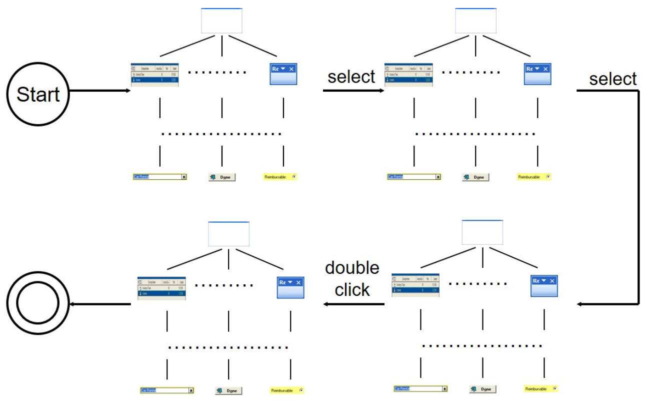

Fig. 2. Representation of a GAP as a state machine where nodes are the tree representations of its GUIs and transitions are labeled with actions on certain GUI objects that trigger corresponding method calls within the GAP.

created, dismiss it, and send an “illegal input” message to the user [20], [21].

B. Background on the GAP State Machine

The representation of a GAP as a state machine is shown in Figure 2. In event-based windowing systems (e.g., Windows, Android), each GAP has the main window (it may be invisible), which is associated with the event processing loop. Closing this window causes the application to exit by sending a corresponding event to the loop. The main window contains other GUI objects of the GAP. A GAP can be represented as a tree, where nodes are GUI objects and edges specify that children objects are contained inside their parents. The root of the tree is the main window, the nodes are container objects, and the leaves of the tree are basic objects [22].

Each GUI object is assigned a category (class) that describes its functionality. For example, in Windows, the basic class of all GUI objects is the class `window`. Some GUI objects serve as containers for other objects, for example, dialog windows, while basic objects (e.g., buttons and edit boxes) cannot contain other objects and are designed to perform some basic functions. Thus, differencing GUI trees includes their topologies and labels of their nodes.

While topologies of GUI trees (i.e., the connectivity among its nodes) may be same, the actual GUIs can differ from one another because of the different types of GUI objects and their attributes. Each GUI object is assigned to a category (class) that describes its functionality. For example, in Windows, the basic class of all GUI objects is the class `window`. Some GUI objects serve as containers for other objects, for example, dialog windows, while basic objects (e.g., buttons and edit boxes) cannot contain other objects and are designed to perform some basic functions.

C. Empirical Study

In our previous work, we studied a problem of maintaining and evolving GUI-directed test scripts that are broken when GUIs are modified between successive releases of the GAP [8]. We conducted an experiment with 45 participants – professional test engineers from Accenture – to evaluate how well these participants can find failures in test scripts when running against the new version of the GAP. One key result of this experiment is that participants found it very difficult to match GUIs manually – it took them from five to twenty minutes to find all differences between GUIs that contain fewer than 30 GUI objects. Moreover, for GUIs that contained superficial changes that did not affect test scripts (e.g., a change in the background colors), it took even longer on average to conclude what changes were made to the GUIs [23].

As part of the preliminary empirical study, we conducted a large-scale survey in Accenture that involved over 2,000 GUI testing and project estimation professionals across six delivery centers in the Americas, Europe, Asia and Oceania. We learned that GUI differencing is a manual and laborious exercise where stakeholders analyze matches between all pairs of GUI objects and select ones that they deem unmatched. We partially automated this approach in our GUI differencing algorithm as part of our previous work where we computed similarities between all pairs of GUI objects and selected matches that were ranked above some user-defined threshold value [8]. Unfortunately, the precision of this algorithm was as low as 28% for some GUIs, even though the participants of the experimental study praised it, since it automated a very difficult task and improved their productivity.

D. Using GAP Source Code To Difference GUIs

Differencing GUIs can be done by comparing the source code of their corresponding GAPs. However, there are two fundamental limitations to code-based matching GUIs.

First, source code of GAPs is not always available especially in black-box testing [24], and GUI testing is inherently black-box since operations are performed on GUI objects rather than objects in the source code, which are instances of classes. Currently, testing is often outsourced to external organizations, and the source code is not shared with these organizations for many reasons. Thus, testing organizations must match GUIs and accessing source code is not an option.

Second, even if the source code is available, there are limitations that render approaches of matching GUIs using source code ineffective. Consider a situation when GUI objects are created using the API call `CreateWindow`, which is used in a large number of Windows GAPs. This API call takes a number of parameter variables including a string variable that holds the value of the type of the GUI object. The value of this variable is often known only at runtime, making it impossible to derive GUI from the source code.

In addition, deriving GUIs from the source code depends on knowing the precise semantics of API calls that create and manipulate GUI objects (e.g., `CreateWindow`), building appropriate parsers and analyzers for languages which are used to create GUI applications, and developing *Integration Development Environment (IDE)*-specific tools that extract GUIs from IDE GUI resource repositories. The number of tuples is measured in tens of thousands in the Cartesian product of API calls $\times$ programming languages $\times$ IDEs. This large number of combinations makes it difficult to come up with an approach that would work with source codebases of different GUI applications. In general, automatically deriving high-level guidance like predicting bugs from the source code is shown to be largely ineffective [25].

E. The Problem Statement

Our goal is to difference GUIs with a high degree of automation and precision. That is, our goal is to compute a function that maps objects from one GUI to their corresponding objects on a different GUI. In this paper, we are interested in mapping each object in one GUI correctly to at most one object of the other GUI or mark this object as deleted if no appropriate mapping exists. It is a difficult problem, since GUIs can be modified in many different ways. Consider a situation where one node in a GUI tree is a container object that contains some other child container object that itself contains a subtree of GUI objects. Suppose that the child container object is deleted and the subtree is assigned to the first container object, i.e., it is moved up the tree by one node. In addition, suppose that types and attributes of some GUI objects in the subtree are modified, e.g., one list object is converted to a combo box and a style of some other object is changed. Even in this simple case, it is difficult to determine precise mappings between the GUIs, since a brute-force tree matching algorithm marks the subtree in the original GUI as deleted and as newly added in the modified GUI.

In general, it is not possible to develop a sound and complete approach for automatically recovering mappings between objects of arbitrary GUIs. An approach is *sound* when objects from one GUI are mapped to objects from a different GUI correctly or not mapped at all. False mappings (i.e., mapping GUI objects incorrectly) are not produced by a sound approach. An approach for computing a GUI mapping function is complete if it recovers mappings for all GUI objects. While a sound and complete approach for automatic differencing GUIs is desirable, it is in general an undecidable problem.

We want to design an automatic approach that replaces a human-driven manual and laborious procedure for recovering mappings between objects of different GUIs with a high precision. That is, our approach should automate the process of searching for patterns in the layouts, the attributes and the labels of GUI objects, and use detected patterns to match objects in different GUIs thus computing a function. Our approach should be lightweight and it should fit into a software development process without introducing additional operations for programmers.

A high degree of automation means that our approach will not impose additional requirements on programmers to introduce additional annotation or to write code in order to march GUIs. Consider a state-of-the-art GUI matching tool ScriptAssure[3] by IBM Corporation as part of the Rational Suite. ScriptAssure works by requiring programmers to assign a smaller number (weight) to the value of an attribute of GUI object and it works mostly when changes are very small across two GUIs (e.g., the background color of an edit box is different). It is needless to say that when changes to GUI are bigger (e.g., a list box is changed to a combo box), ScriptAssure no longer works. The process of using ScriptAssure is manual, with test engineers manually assigning weights to different properties of GUI objects and hoping that mappings between these objects can be determined automatically. ScriptAssure does not handle even simple cases well – when positions of two GUI objects are switched, it takes a significant effort to make ScriptAssure define the correct mappings. Thus, our goal is to show that it is possible to difference GUIs with a high degree of automation and precision.

[3]http://www-01.ibm.com/software/awdtools/tester/functional/sa.html

III. Solution

In this section, we formalize the problem of differencing GUIs, explain how we extract GUI models from GAPs, provide a baseline algorithm for GUI differencing and describe our solution that is based on tree edit distance algorithms to difference GUIs.

A. Extracting GUI Trees From Running GAPs

Recall that our goal is to obtain GUI trees in a non-intrusive, platform and language-independent way. To do that, we use a set of generic interfaces of the accessibility layer, with which *GUI DifferEntiator (GUIDE)* traverses the GAP tree starting from the root, i.e., the main window. Once at some GUI object, GUIDE extracts values of its properties (i.e., the name of the object, its role, its type, its coordinates, values, its type, its position in the tree hierarchy, and styles (e.g., color, type of the border, modality). These properties are used to compare trees in order to find modified GUI objects.

Importantly, to obtain certain properties, it is necessary to determine how GUI objects behave (e.g., if a GUI object accepts user input). However, GUI objects exist in the protected process space of the running GAP and to investigate their behaviors automatically, it is necessary to initiate events (i.e., to create data structures that represent some actions on GUI objects) that simulate interactions with users. Since accessibility layers are supported by their respective vendors, using interfaces that are exposed by accessibility layers is a legitimate and accepted way to control and manipulate GAPs. When a target GAP is started, the accessibility layer libraries are loaded in the process space of this applications and registers addresses of callback methods that should be invoked in response to specified events. Since accessibility libraries "live" in the process spaces of GAPs, they can affect every aspect of execution of these GAPs. For each GUI object, accessibility layers have a range of events to test its properties. Results of these tests are recorded in the extracted GUI tree.

B. Computing Similarity Between Trees

Our previous work on differencing GUIs resulted in a baseline GUI differencing algorithm that computes the similarity between all pairs of GUI objects on two GUIs [26]. We improve on it algorithm in this paper. A main idea of this algorithm is to compute the matching score between each GUI object in the tree Γ_n and all objects of the tree Γ_{n+1}. The highest score determines the winning mappings that are added to the mapping set. This algorithm was created to imitate how stakeholders difference GUIs manually by selecting an object in the GUI tree Γ_n and determining its most similar GUI objects in the tree Γ_{n+1}. The similarity score was computed by taking a normalized linear sum between the values of the corresponding matching properties of GUI objects. In case of coordinates, matching is approximate, given some distance threshold between locations of the GUI objects. We used an early variant of this algorithm to automate GUI matches as part of our tool REST for maintaining and evolving GUI-directed test scripts [8]. Even though the accuracy of this tool was somewhat low, an average of $\approx 30\%$, users liked it, since it automated a difficult and important task. We use an improved version of this algorithm as a baseline GUI matching tool in this paper.

A comparison algorithm called `BaselineDiffGuiTree` for computing mappings between GUI objects of GUI trees is shown in Algorithm 1. This algorithm takes as its input the GUI trees Γ_n and Γ_{n+1} for two successive releases of the same GAP and computes the mapping set μ that contains relations between GUI objects of these trees along with their match score σ, $((\theta,\rho)\in\mu,\sigma),\theta\in\Gamma_n,\rho\in\Gamma_{n+1},0\leq\sigma\leq 1$. The match score σ is computed as a weighted sum of a proximity criteria between values of the properties of GUI objects. When $\sigma = 1$, the objects are identical, and when $\sigma < 1$ the match is partial.

Algorithm 1 The BaselineDiffGuiTree algorithm.

```
BaselineDiffGuiTrees( $\Gamma_n$, $\Gamma_{n+1}$ )
$\mu \mapsto \emptyset, \sigma \mapsto 0$
repeat
  for all $\theta \in \Gamma_n$ s.t. $\forall \nu \in \Gamma_{n+1}.((\theta,\nu)_f \notin \mu)$ do
    for all $\rho \in \Gamma_{n+1}$ s.t. $\forall \omega \in \Gamma_n.((\omega,\rho)_f \notin \mu)$ do
      $\sigma \mapsto$ ComputeScore$(\theta,\rho)$
      if $\neg(\exists\kappa \in \Gamma_n.((\theta,\kappa)\in\mu) \vee \exists\lambda \in \Gamma_{n+1}.((\lambda,\rho)\in\mu))$
      then
        if $\sigma > \varepsilon$ then
          $\mu \mapsto \mu \cup ((\theta,\rho),\sigma)$
        end if
      else if $\exists\kappa \in \Gamma_n.((\theta,\kappa)\in\mu) \wedge \neg\exists\lambda \in \Gamma_{n+1}.((\lambda,\rho)\in\mu)$
      then
        if GetScore$(\theta,\kappa) < \sigma$ then
          $\mu \mapsto \mu \backslash (\theta,\kappa)$
          $\mu \mapsto \mu \cup ((\theta,\rho),\sigma)$
        end if
      else if $\neg\exists\kappa \in \Gamma_n.((\theta,\kappa)\in\mu) \wedge \exists\lambda \in \Gamma_{n+1}.((\lambda,\rho)\in\mu)$
      then
        if GetScore$(\lambda,\rho) < \sigma$ then
          $\mu \mapsto \mu \backslash (\lambda,\rho)$
          $\mu \mapsto \mu \cup ((\theta,\rho),\sigma)$
        end if
      else
        {Both GUI objects $\theta$ and $\rho$ are already mapped
        to some other objects}
        if GetScore$(\theta,\kappa) < \sigma \wedge$ GetScore$(\lambda,\rho) < \sigma$
        then
          $\mu \mapsto \mu \backslash (\lambda,\rho)$
          $\mu \mapsto \mu \cup ((\theta,\rho),\sigma)$
        end if
      end if
    end for
  end for
until $\mu$ is not changed
```

A main idea of this algorithm is to compute the matching score σ between each GUI object in the tree Γ_n and all objects of the tree Γ_{n+1}. The highest score determines the winning mapping that is added to the set μ. However, it is possible that one of the considered objects, or both of them can already be mapped to some other objects. In this case we compare the scores between old and new mappings, and if the new mapping has a higher score, then we delete the old mapping and add the new one. We keep repeating this operation until there are no more changes to the set μ. The algorithm converges after m passes where m is the number of nodes in the tree Γ_n. The proof is by induction on the size of the tree.

`DiffGuiTree` ignores fixed mappings between GUI objects that the user determines using the front end of Rest. We denote fixed mappings between GUI objects θ and ρ as $(\theta,\rho)_f \in \mu$. Functions `ComputeScore` and `GetScore` compute and retrieve the matching scores σ correspondingly.

C. Differencing GUIs Using Tree Edit Distance Algorithms

Edit distance is a popular approach for quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other [27, pp 107-111]. String distance metrics are considered as a precursor to the tree edit distance – e.g., a Levenshtein distance is computed for a simple set of operations with their costs that included copy/map a character from one string to the other with the cost zero and delete or insert a character with the cost one [28]. Algorithms based on edit distances are used in different areas, e.g., for automatic spelling correction in natural language processing and in bioinformatics to quantify the similarity of DNA molecules.

A comprehensive survey on using *tree edit distance (TED)* algorithms to compare trees states that these algorithms compute the edit distance and a sequence of edit operations between two trees that minimize the cost function defined for these operations [29]. TED includes tree alignment and tree inclusion distances. *Tree alignment* is performed by inserting empty nodes in the GUI tree, as adjusted for our definition of the GUI tree, so that the trees Γ_n and Γ_{n+1} become isomorphic if labels are disregarded. The cost of the alignment is the sum of the costs of all label mismatches in the overlapped trees. *Tree inclusion* address a problem of determining if the tree Γ_n is included in Γ_{n+1}. Edit distance algorithms are based on relabeling, inserting and deletion of nodes, the same key operations that stakeholders use to modify GUIs. We use tree edit distance algorithms to difference GUIs by computing the minimum cost solution of transforming one tree into the other by a sequence of elementary operations consisting of deleting and relabeling existing GUI objects, as well as inserting new objects.

Tai introduced a TED criterion in 1979 for computing a distance between two trees [30] and Shasha and Zhang extended this work by introducing a left-to-right order comparison algorithm for trees [31]. There can be multiple Tai mappings between two trees, and Shasha and Zhang's algorithm selects a mapping that has the minimum edit distance cost with the worst running time of $O(n^4)$, where n is the number of nodes in the tree. A key feature of the Shasha and Zhang algorithm is that the two rightmost roots of the trees must be matched recursively and the edit distance and the cost is computed for matching. Alternatively, the Klein's algorithm recurses on the left root if the size of the leftmost tree is smaller than the rightmost tree and on the right root otherwise with the worst complexity $O(n^3 \log n)$[32].

These results are improved by the algorithm RTED that gives the worst case complexity $O(n^3)$ [33]. RTED is based on an adaptive recursive decomposition strategy that uses the divide-and-conquer approach. A key idea behind RTED is that there are certain subtrees for which it is worthwhile to choose the direction according to the currently larger tree branch, while for other subtrees we had better keep choosing the direction according to the originally larger branches.

IV. Experiment

In this section, we pose research questions and describe the methodology and the setup of our experiment for evaluating GUIDE on GAPs.

A. Research Questions

We pose the following research questions (RQs).

RQ1: how effective tree-edit algorithms are when compared to the baseline comparison approach?
RQ2: how efficient tree-edit algorithms are for practical uses on large-scale GUIs?
RQ3: are performance characteristics of tree-edit algorithms affected by the ratios of changed GUI objects between compared GUIs?

A rationale for RQ1 is that tree-edit algorithms are more accurate than the baseline brute-force all-node comparison approach that we described in Section III-B [26]. With RQ2, our goal is to determine requirements for resources and running times for GUIDE when used on larger GUIs. Excessive resource demands may render GUIDE impractical. Finally, we believe that the effectiveness of tree comparison algorithms is reduced when the number of changes between two GUIs is excessive. A rationale for RQ3 is to investigate how the differencing algorithms perform when the number of GUI objects increases that are different between two GUIs.

B. Methodology

A key driver of our experimental methodology is to cover a wide range and complexities of GUIs as well as various combinations of changes that can be performed on their structures. Our goal is to determine how well tree edit algorithms difference GUI trees, so that we can determine their suitability for different software engineering tasks. An experiment is straightforward – the input is GUI trees Γ_n and Γ_{n+1}, where Γ_{n+1} is a modified version of Γ_n. A golden set of differences between Γ_n and Γ_{n+1} is known in advance. Once GUIDE runs a differencing algorithm on these trees, it produces mappings between the GUI objects of these trees that are compared to the golden set. Ideally, this experiment should be conducted on various GAPs of different sizes from small to large with different changes between versions of GUIs, so that we can determine the benefits and limitations of tree differencing algorithms.

Unfortunately, it is very difficult to obtain and run a large number of GAPs with nontrivial GUIs and with many different changes across versions of these GUIs, not to mention the difficulty of obtaining golden sets. Large-scale GAPs that contain hundreds of GUI objects on each GUI are cumbersome to run, since they are built on top of various packages (e.g., SAP, Peoplesoft) and they require sophisticated platforms and databases. Moreover, compositions of GUI objects can be specific to software projects or to a company that created the GAP, so that the population of subject GAPs should be highly diversified in order to avoid a bias.

Writing benchmark GAPs from scratch requires a lot of manual effort, not to mention that a significant bias and human error can be introduced [34]. In addition, selecting commercial applications as benchmarks negatively affects reproducibility of results, which is a cornerstone of the scientific method [35], since commercial benchmarks cannot be easily shared among organizations and companies for legal reasons and trade-secret protection. For example, Accenture Confidentiality Policy (item 69) [4] states that source code, which is generated by the company and relates to its business, research and development activities, clients or other business partners, or employees are considered confidential information. Other companies have similar policies.

Ideally, users should be able to easily generate benchmark GAPs with desired properties. This idea has already been used successfully in testing relational database engines, where complex *Structured Query Language (SQL)* statements are generated using a random SQL statement generator [36]. Suppose there is a claim that a relational database engine performs better at certain aspects of SQL optimization than some other engines. The best way to evaluate this claim is to create complex SQL statements as benchmarks for this evaluation in a way that these statements have desired properties that are specific to these aspects of SQL optimization, for example, complicated nested SQL statements that contain multiple joins. Since the meaning of SQL statements does not matter for

[4] https://policies.accenture.com/Pages/0001-0100/0069.aspx

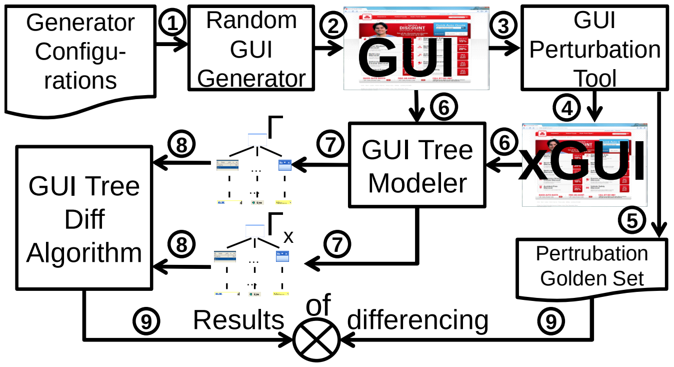

Fig. 3. An experimental flow for evaluating GUIDE.

performance evaluation, this generator creates semantically meaningless but syntactically correct SQL statements thereby enabling users to automatically create low-cost benchmarks with significantly reduced bias. We used a random application generation approach in our previous work [37], [38], [39].

C. Random GUI Benchmark Generation

In this paper, we use an idea of random tree generation [40], [37], [39] In GUIDE, we use this formal definition to extract GUI trees from running GAPs. An opposite use of this definition is to generate GUI trees for different labels and nodes, where each node and label is assigned the probability with which it is instantiated in a GUI. These trees are called *stochastic*, and they are widely used in natural language processing, speech recognition, information retrieval [41], and also in generating SQL statements for testing database engines [36]. We use a *stochastic GUI tree model* to generate large random GAPs.

Random GUI trees are constructed based on the stochastic tree model, and the construction process can be described as follows. Starting with the top root node of the tree, each node and edges that connect it to other nodes are recursively generated. Nodes are chosen based on their probabilities of appearing in the GUI tree. Label and attributes are replaced with randomly generated identifiers and values that preserve rules of the given GUI framework. Termination conditions for this process of generating GUIs include the limit on the size of the tree or selected complexity metrics.

D. The Setup of Experiments

An experimental setup for evaluating GUIDE is shown in Figure 3. The Random GUI Generator takes the GUI configuration `(1)` that describes the ranges of GUI objects on the GUI and various constraints and the probabilities that the Generator uses `(2)` to output the GUI. The next step involves the GUI Perturbation tool that `(3)` modifies randomly chosen GUI objects on the generated GUI `(4)` thus producing xGUI as well as `(5)` the Perturbation Golden Set that records changes to the GUI. The GUI Tree Modeler `(6)` takes the GUI and the xGUI as its input and `(7)` computes GUI tree models Γ and Γ_x. After that these GUI trees are `(8)` inputted to the GUI Tree Differencing algorithm that matches these GUI trees and `(9)` produces differences that are compared with the golden set. The process repeats until the termination conditions for the experiment are satisfied (i.e., the size of the explored GUI set and the length of the experiment).

Configuration parameters for the random GUI generator include ranges for the numbers of GUI objects per GUI, the depth of the generated GUI tree, maximum and minimum number of branches for tree nodes, probabilities for different types of GUI objects with which they appear of a GUI. The latter allows us to generate balanced GUIs where more objects display information (e.g., list and combo boxes) and fewer objects are created for user interactions (e.g., menus and buttons). Configuration parameters for the perturbation tool include the maximum and minimum percentages of the total number of GUI objects which should be modified and types of modifications allowed. For each configuration set we generate 1,000 different pairs of GUIs, so that we obtain comprehensive results based on the diversity of subject GUIs. We carry out experiments in the cloud infrastructure where we execute experiments in parallel in 500 virtual machines. The total experimentation time is over 18 hours.

E. Open-Source Subject Applications

We selected three open source subject GAPs based on the following criteria: easy-to-understand domain, limited size of GUI (less than 200 GUI objects total), and two successive releases of GAPs with modified GUI objects. `Twister` (versions 2.0 and 3.0.5) is a real-time stock quote downloading programming environment that allows users to write programs that download stock quotes[5]. `mRemote` (versions 1.0 and 1.35) enables users to manage remote connections in a single place by supporting various protocols (e.g., SSH, Telnet, and HTTP/S)[6]. Finally, `Budget Tracker` (versions 1.06 and 2.1) is a program for tracking budget categories, budget planning for each month and keeping track of expenses[7]. These applications are nontrivial and they are highly ranked in Sourceforge. They are Windows applications, and it is important that GUIDE is used without any modifications both on the browser-based generated Ajax applications and on Windows open-source ones.

F. Hypotheses

We introduce the following null hypotheses to evaluate how close the means are for the F-measures for control and treatment groups. Unless we specify otherwise, in the treatment group tree-edit algorithms are used, and in the control group the baseline differencing algorithm is used. We seek to evaluate the following hypotheses at a 0.05 level of significance.

[5]http://sourceforge.net/projects/itwister/
[6]http://sourceforge.net/projects/mremote/
[7]http://sourceforge.net/projects/budgettracker/

H_0^A The null hypothesis A is that there are no differences in the F- and G-measures between the baseline and tree-edit algorithms where the percentages of changed GUI objects are the same.

H_0^B The null hypothesis B is that there are no differences in the F- and G-measures between the baseline and tree-edit algorithms across categories with different percentages of changed GUI objects.

Corresponding alternative hypotheses to H_0 is that there are statistically significant differences in the F- and G-measures. We do not state them explicitly for brevity.

G. Variables

Independent variables for the experiment include the range of GUI objects per GUI, the maximum percentage of the GUI objects that can be perturbed on a GUI page and the tree comparison algorithm that we use to evaluate. Each pair of GUIs has the source GUI and the target GUI where some GUI objects are modified using the perturbation tool that is applied to the source GUI. Dependent variables include the following. True negatives, TN, gives the number of GUI objects that are perturbed and correctly identified by a tree comparison algorithm as not mapped to any GUI object on the target GUI. True positives, TP, gives the number of GUI objects on the source GUI that are correctly mapped to their counterparts on the target GUI. False negatives, FN, is the number of GUI objects that a comparison algorithm identifies as modified while in fact they were not modified. Conversely, false positives, FP, is the number of GUI objects that a comparison algorithm identifies as mapped to their counterparts on the target GUI while in fact they were modified.

Precision, P, and recall, R, are main ratios with which we measure the qualities of the comparison algorithms. The precision ratio is computed as $P = \frac{TP}{TP+FP}$, and the recall ration is computed as $R = \frac{TP}{TP+FN}$. The precision ratio shows how mistaken the comparison algorithms are, and the recall is the ratio of correctly recovered mappings between GUI objects in source and target GUIs.

The idea behind computing the precision and recall is to evaluate the difference between TPs and FPs. If all identified failures are in fact real failures and not FPs, i.e., FP=0, then precision=1. If all identified mappings are FP, then $P = 0$ and $R = 0$. The ratio recall is analogous to the recall parameter in information retrieval, which is the ratio of the number of relevant documents retrieved by a search divided by the total number of existing relevant documents, and the ration precision is analogous to the definition of precision in information retrieval, which is the number of relevant documents retrieved to the total number of documents retrieved by the search.

Other dependent variables include the specificity or true negative value, computed as $S = \frac{TN}{TN+FP}$ measures the ratio of true negatives which are correctly identified w.r.t. FPs. Negative predictive value is measured as $NPV = \frac{TN}{TN+FN}$ and it shows the ratio that GUI objects that are determined to be modified by a comparison algorithm were truly modified. Finally, F and G measures are combined characteristics of precision and recall; F-measure gives the harmonic mean of precision and recall and G-measure is the geometric mean of precision and recall.

H. Threats to Validity

A threat to the validity of this experimental evaluation is that our evaluation is performed on artificially generated GUIs. Real-world GUI applications may have properties that are different from artificially generated ones, however, with our random generation tool we minimized possible differences and concentrated on pushing the boundaries by generating very complex GUIs. To be on the safe side, we used three open-source GAPs and the results of our evaluation of GUIDE are in agreement for generated GUIs and the GUIs of the open-source applications.

The other threat to validity is that we experimented using only two GUI framework, specifically Ajax and Microsoft Windows and only MSAA from a list of available accessibility technologies. It is possible that there are less popular GUI frameworks and accessibility technologies that use different mechanisms for event delivery and interactions with GUI objects. To the best of our knowledge, most popular GUI frameworks that we experimented with use the same mechanisms qualitatively that we described in this paper.

Next, a threat to validity is that some complex topologies for GUI trees can be missed in our evaluation. We counter this threat by generating a very large number of GUIs, specifically 5,000. It is highly unlikely that a human experimenter can manually create a GUI whose topology is not covered in the experiments that we described in this paper.

In addition, GAPs can contain GUI objects that are not designed to be accessible, that is to comply with the accessibility standards. In this case, GUIDE will not work, but considering that different governments mandate GAPs to comply with accessibility standards, it means that GAPs that will not comply with these standards will not be commercially available, since respective governments will prevent sales. Thus, GUIDE can still be used for a majority of GAPs that comply with accessibility regulations.

Finally, we do not consider dynamically modified GUIs, where interacting with one GUI object may change the layout of the entire GUI [42]. Such GAPs exist, but we believe they are rare, since changing GUI layout based on user actions makes it difficult for users to work with such GAPs. Yet, since GUIDE models the GUIs of running GAPs, it is possible to apply it to dynamically created GUIs. Extending GUIDE to support various aspects of dynamically modified GUIs is a subject of future work.

V. Results

Experimental results are shown in Table I. The first column, GUI objects lists five categories for the ranges of GUI objects for the generated GUIs. The first range is 30–100 starts at 30 and goes to 100 objects per a generated GUI. The second range is 100–200 and so on up to 500 objects per GUI. For each range, we generated 1,000 GUIs. For each GUI

TABLE I

Results of experiments on generated subject GUIs with different GUI tree differencing algorithms (Alg): Baseline (Base), Shasha/Zhang (Sh-Zhang), Klein and RTED. The first column, Objects, specifies the ranges of GUI objects in the generated GUIs. For each range, 1,000 pairs of GUIs are generated with different percentages of GUI objects modified for each pair – the percentage is specified in the second column, M. The next ten give the average values for each algorithm for true negatives (TN), false negatives (TN), true positives (TP), false positives (FP), precision (P), recall (R), specificity (S), negative predictive value (NPV), the G measure and the F measure. We also give the minimum and maximum execution time for the algorithm in seconds and its maximum memory usage in Mb.

GUI Obj	M	Alg	TN_{avg}	FN_{avg}	TP_{avg}	FP_{avg}	P_{avg}	R_{avg}	S_{avg}	NPV_{avg}	G_{avg}	F_{avg}	T_{min}	T_{max}	M
30-100	5%	Base	2.66	14.85	12.27	21.14	0.37	0.47	0.18	0.28	0.39	0.36	33	35	549
		Sh-Zha	2.67	14.96	22.27	11.05	0.66	0.60	0.29	0.28	0.61	0.59	658	799	3103
		Klein	2.77	15.82	22.42	10.43	0.67	0.58	0.32	0.28	0.60	0.58	523	598	1292
		RTED	2.74	15.15	22.41	10.89	0.67	0.59	0.31	0.28	0.60	0.58	628	682	1529
	25%	Base	15.94	18.70	8.55	18.03	0.33	0.35	0.53	0.53	0.31	0.29	103	174	302
		Sh-Zha	15.72	19.19	16.38	9.68	0.64	0.48	0.68	0.53	0.53	0.51	350	843	2690
		Klein	15.83	18.42	16.40	10.72	0.61	0.48	0.66	0.53	0.52	0.49	491	519	1918
		RTED	15.61	18.20	16.01	9.74	0.61	0.48	0.68	0.54	0.52	0.49	592	639	1308
101-200	5%	Base	7.01	34.28	28.94	49.65	0.37	0.46	0.20	0.30	0.38	0.36	184	554	626
		Sh-Zha	6.97	34.47	53.23	25.44	0.67	0.61	0.33	0.29	0.62	0.60	548	645	4171
		Klein	7.06	34.92	54.14	25.81	0.68	0.60	0.34	0.28	0.61	0.59	1885	1934	2609
		RTED	7.01	34.73	52.90	25.07	0.67	0.59	0.34	0.28	0.61	0.59	1685	1693	2657
	25%	Base	37.21	44.25	18.75	44.22	0.30	0.32	0.52	0.53	0.29	0.27	184	202	224
		Sh-Zha	36.73	43.29	35.50	24.09	0.60	0.46	0.67	0.53	0.50	0.48	811	945	6751
		Klein	37.25	43.13	37.50	23.32	0.61	0.48	0.68	0.53	0.52	0.49	1730	2046	2556
		RTED	37.45	45.72	37.15	23.37	0.61	0.46	0.68	0.52	0.51	0.48	1853	2008	3870
201-300	5%	Base	12.01	59.49	42.59	88.92	0.33	0.44	0.19	0.30	0.35	0.32	263	528	671
		Sh-Zha	12.02	57.94	89.78	43.36	0.67	0.60	0.34	0.30	0.61	0.60	3419	3568	5928
		Klein	12.11	60.03	89.25	42.46	0.68	0.60	0.35	0.29	0.61	0.59	3530	4772	5610
		RTED	11.98	58.46	87.48	42.27	0.67	0.60	0.35	0.29	0.61	0.59	3431	4034	5597
	25%	Base	61.53	70.97	34.40	69.77	0.33	0.35	0.53	0.53	0.32	0.30	685	896	681
		Sh-Zha	61.57	70.99	64.60	37.28	0.63	0.48	0.69	0.53	0.53	0.51	3983	4885	5714
		Klein	62.15	75.69	62.34	37.85	0.61	0.46	0.68	0.52	0.51	0.48	3557	3761	4621
		RTED	62.03	71.40	63.25	39.76	0.62	0.48	0.67	0.54	0.52	0.49	3385	3457	5495
301-400	5%	Base	16.99	81.62	53.35	132.09	0.28	0.42	0.17	0.30	0.32	0.29	315	337	857
		Sh-Zha	17.02	81.05	118.05	64.48	0.64	0.58	0.34	0.29	0.59	0.57	6010	6161	5353
		Klein	17.00	81.48	123.12	58.76	0.67	0.60	0.35	0.29	0.61	0.59	5189	5536	7172
		RTED	17.04	82.96	124.20	60.93	0.67	0.59	0.35	0.28	0.61	0.59	5072	7522	6977
	25%	Base	87.58	103.80	34.51	107.93	0.24	0.28	0.50	0.53	0.24	0.22	426	492	1065
		Sh-Zha	86.93	102.42	83.26	57.61	0.59	0.46	0.67	0.53	0.50	0.47	5212	5883	6265
		Klein	87.28	104.09	88.79	56.32	0.61	0.47	0.67	0.52	0.51	0.49	5698	6611	6189
		RTED	86.88	103.41	85.62	54.10	0.61	0.46	0.68	0.53	0.51	0.49	6664	7357	5583
401-500	5%	Base	22.01	102.28	55.85	185.45	0.23	0.39	0.16	0.30	0.28	0.25	112	115	1237
		Sh-Zha	22.05	106.98	171.33	67.22	0.72	0.61	0.38	0.29	0.64	0.62	6434	7407	6817
		Klein	22.06	110.45	156.00	74.44	0.68	0.58	0.36	0.28	0.60	0.58	7628	9268	6822
		RTED	21.96	106.94	158.44	77.86	0.67	0.60	0.34	0.29	0.61	0.59	8752	9954	7703
	25%	Base	112.37	131.59	36.32	153.44	0.19	0.26	0.47	0.53	0.21	0.19	195	894	1680
		Sh-Zha	111.82	134.86	117.49	67.37	0.65	0.47	0.70	0.52	0.53	0.50	7652	9635	7999
		Klein	112.18	132.15	113.86	71.10	0.61	0.47	0.67	0.53	0.52	0.49	6485	6612	6121
		RTED	112.25	130.55	113.18	69.53	0.61	0.47	0.68	0.53	0.52	0.50	6950	9680	6053

we report results for 5% and 25%. We also report running times and memory usages for different experiments in the last three columns of the table. We report mostly averages for dependent variables. Spreadsheets with full results are available for download from the project's website.

A key result is that the precision is significantly lower for the baseline algorithm when compared with the precision values for the tree-edit algorithms. The same statement holds true for the F- and G-measures. The second result is that the precision and recall of all algorithms are decreased when the number of changed objected between GUIs is increased. Our observation is that this number varies significantly depending on the particular GUI, however, we observe that when the ratio of changed objects between two GUIs is over 15% the precisions of the differencing algorithms decrease significantly by more than 30%.

The results of ANOVA confirm that there are large differences for F- and G-measures between the baseline algorithm and tree-edit algorithms with $\frac{F}{Fcrit}$ as > 31 with p-value=3.8E-18 that shows strong statistical significance. The average F- and G-measures for the baseline approach are more than 80% lower than for tree-edit distance algorithms. Additional t-tests confirmed with high degrees of statistical significance that tree-edit distance algorithms outperformed the baseline approach. However, as we analyzed the results among only tree-edit distance algorithms separately, we discovered that their precisions do not differ with strong statistical significance. Results of the experiments with three open source subject applications are within the same ranges that we show for

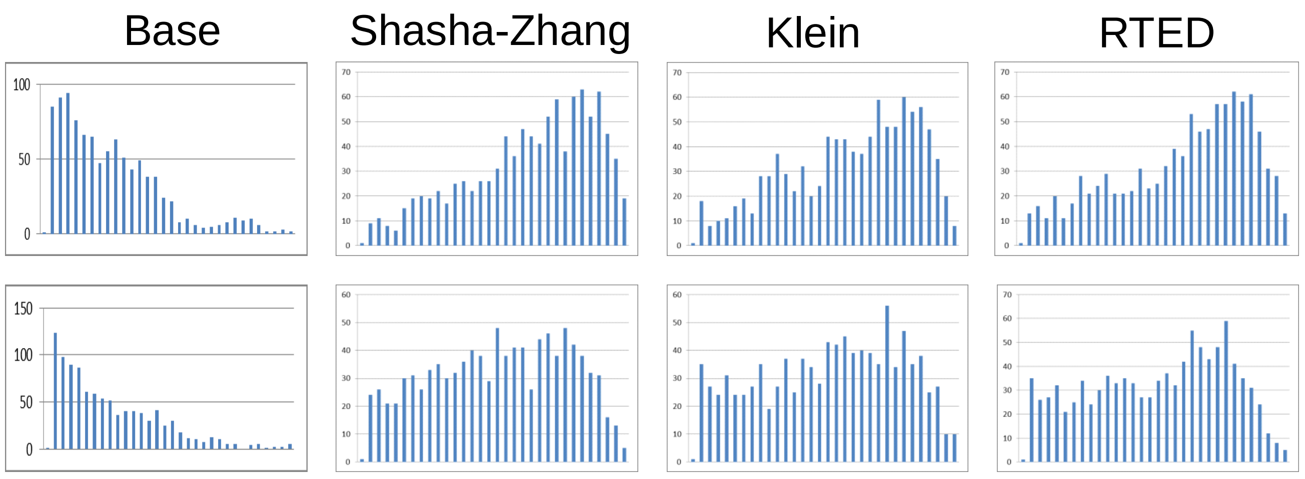

Fig. 4. Histograms for F-measures for the algorithms for GUIs with 500 objects. The upper row shows histograms for 5% perturbation rate and the bottom row shows for 25% perturbation rate. The labels on the top of each column of graphs show the type of the differencing algorithm for which the histograms are generated. It is easy to see that the histogram density is higher for the smaller values of the F-measure for the BASE algorithm and much higher for the tree-edit algorithms meaning that the precision of the latter is higher. Also, one can see the difference between histogram in each column for different perturbation rates. As the perturbation rate is higher, i.e., 25% vs 5% in the lower row, the density of the F-measure values shifts towards the center, i.e., towards the lower values when compared with the corresponding histogram for 5% perturbation rate in the upper row.

artificially generated GUIs, thus addressing a threat to validity that artificially generated GUIs may lead to skewed results.

To answer RQ1 we determine if the F- and G-measures for the experiments using tree-edit algorithms are statistically higher than the corresponding F- and G- measures for the experiments with the baseline approach. We test the hypothesis H_0^A to determine if there are significant differences between performances of these algorithms within each category of M, i.e., separately within the 5% and 25% categories. Based on our results, **we affirmatively answer RQ1 and we reject the hypothesis H_0^A by stating that tree-edit algorithms are more effective when compared to the baseline approach**.

To answer RQ2 we estimate the running time and memory usages for tree-edit algorithms. We can see from Table I that running times reach approximately three hours for the largest GUIs with the memory requirements reaching eight gigabytes. These time and space requirements, while large, are acceptable for modern computers. Moreover, multiple GUIs can be compared in parallel in multiple virtual machines in a cloud environment, thus making this task practical. Thus, **we affirmatively answer RQ2 that tree-edit algorithms are efficient and good for practical uses on large-scale GUIs**.

Finally, to answer RQ3 we determine if there is a significant difference between the values of F- and G-measures for all algorithms between the categories of modified GUI objects, i.e., between 5% and 25% categories. Eight histograms for F-measures for different experiments are shown in Figure 4, two for each tree differencing algorithm. The upper row contains histograms for the 5% change category and the bottom row contains histograms for 25% category. We observe that the density of F-measures shifts towards the lower values (i.e., to the left) for the bottom row. That is, the overall precision of the algorithms is lower when the number of changed objects between GUIs is greater. To quantify an answer to this conjecture, we test the hypothesis H_0^B to determine if there are significant differences between performances of these algorithms across the modification percentage categories. The results of t-tests for the directionality of means confirm that there are significant differences between mean values for four tree differencing algorithms that we use in this paper with the $p < 10^5$. Thus, **we affirmatively answer RQ3 and we reject the null hypothesis H_0^B and accept the alternative hypothesis that states that there are significant differences in the F- and G-measures between the baseline and tree-edit algorithms across categories with different percentages of changed GUI objects**.

VI. Related Work

Differencing GUIs in GAPs present special challenges to regression testing because the input-output mapping does not remain constant across successive versions of the software [43][44]. Numerous techniques have been proposed to automate regression testing by differencing GAPs while relying on information obtained from the modifications made to the source code. Some of the popular regression testing techniques include analyzing the program's control-flow structure [45], analyzing changes in functions, types, variables, and macro definitions [46][47], using def-use chains [48], constructing procedure dependence graphs [49][50], and analyzing code and class hierarchy for object-oriented programs [51][52]. In addition, regression testing technique for GUIs (GUITAR) repairs test cases that have become unusable for the modified GUIs [9]. These techniques are not directly applicable to

GUI differencing, since regression information is derived from changes made to the source code.

When it comes to extracting information from GAPs and their GUI elements, the term *screen-scraping* summarily describes various techniques for automating user interfaces [53][54]. Macro recorders use this technique by recording the user's mouse movements and keystrokes, then playing them back by inserting simulated mouse and keyboard events in the system queue [55]. Screen-scraping has multiple advantages over binary rewriting and code patching techniques as it does not require any modifications to the underlying computing platforms or applications' source and executable code. GUIDE uses some aspects of screen-scraping, however, it differs from other screen-scraping techniques since it does not depend on parsing a scripting language that describes the GUI, and therefore it is more generic and uniform.

In model-based GUI development and testing, building high-level models of GAPs before applying algorithms that construct test cases for evolved GAPs [56], [57], [58], [59]. We consider GUIDE complementary to modeling GUIs since it could enhance GUIDE and improve its precision and usability by utilizing richer models.

An automated technique called FMAP matches features of web application in cross-platform development [60]. X-pert is a different tool by the same authors for identification of cross-browser issues [61]. Unlike FMAP that captures the request-response pairs sent by the browser and X-pert that is based on an assumption of a single platform, GUIDE uses a generic accessibility technique with a novel use of tree-edit algorithms and it does not require stakeholders to collect and analyze execution traces of the applications.

VII. Conclusions

We offer a novel approach for differencing GUIs that combines tree edit distance measure algorithms with accessibility technologies for obtaining GUI models in a non-intrusive, platform and language-independent way, and it does not require the source code of GAPs. We developed a tool called *GUI DifferEntiator (GUIDE)* that allows users to difference GUIs of running GAPs. To evaluate GUIDE, we created an experimental platform that generates random GUIs with controlled differentials among them that serve as oracles, enables researchers to plug-and-play different differencing algorithms, and automatically runs experiments and produces results. We evaluated GUIDE on 5,000 pairs of generated GUIs as well as three open-source ones. The results of our evaluation and user experience suggest that our approach can find differences between GUIs with a high degree of automation and precision.

Acknowledgments

We thank many CS graduate students at UIC who contributed to GUIDE as part of their work on their MS projects and theses. We warmly thank anonymous reviewers for their comments and suggestions that helped us to improve the quality of this paper. This work is supported by NSF EAGER-1650000, NSF CCF-1615563, NSF I-Corps-1547597, Microsoft, Amazon, Accenture, and Grammatech. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, September 2004.

[2] M. Fewster and D. Graham, *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, September 1999.

[3] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *FOSE '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–103.

[4] M. Grechanik, Q. Xie, and C. Fu, "Creating GUI testing tools using accessibility technologies," in *ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, 2009, pp. 243–250.

[5] K. M. Conroy, M. Grechanik, M. Hellige, E. S. Liongosari, and Q. Xie, "Automatic test generation from GUI applications for testing web services," in *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, 2007, pp. 345–354. [Online]. Available: https://doi.org/10.1109/ICSM.2007.4362647

[6] C. Kaner, "Improving the maintainability of automated test suites," *Software QA*, vol. 4, no. 4, 1997.

[7] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *ICSE '05*, New York, NY, USA, 2005, pp. 571–579.

[8] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *ICSE*, 2009, pp. 408–418.

[9] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," in *Proceedings of the ESEC and FSE-11*, Sep. 2003, pp. 118–127.

[10] B. W. Boehm, "A spiral model of software development and enhancement." *IEEE Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[11] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping vs. specifying: A multi-project experiment," in *ICSE '84*. Piscataway, NJ, USA: IEEE Press, 1984, pp. 473–484.

[12] K. Weide, "Worldwide new media market model 1h12 highlights: Internet becomes ever more mobile, ever less pc based." *Technical Report 237459*, Oct. 2012.

[13] A. Michail, "Browsing and searching source code of applications written using a gui framework," ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 327–337.

[14] Q. Xie, M. Grechanik, and C. Fu, "REST: A tool for reducing effort in script-based testing," in *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, 2008, pp. 468–469. [Online]. Available: https://doi.org/10.1109/ICSM.2008.4658108

[15] C. Fu, M. Grechanik, and Q. Xie, "Inferring types of references to GUI objects in test scripts," in *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*, 2009, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICST.2009.12

[16] "Section 508 of the Rehabilitation Act," *http://www.access-board.gov/508.htm*.

[17] J. Mueller, *Accessibility for Everybody: Understanding the Section 508 Accessibility Requirements*. APress L. P., 2003.

[18] K. Harris, "Challenges and solutions for screen reader/i.t. interoperability," *SIGACCESS Access. Comput.*, no. 85, pp. 10–20, Jun. 2006.

[19] M. Dorigo, B. Harriehausen-Mühlbauer, I. Stengel, and P. S. Dowland, "Survey: Improving document accessibility from the blind and visually impaired user's point of view," ser. UAHCI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 129–135.

[20] M. Grechanik, K. M. Conroy, and K. S. Swaminathan, "Creating web services from gui-based applications," in *IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2007, 19-20 June 2007, Newport Beach, California, USA*, 2007, pp. 72–79. [Online]. Available: https://doi.org/10.1109/SOCA.2007.16

[21] M. Grechanik and K. M. Conroy, "Composing integrated systems using gui-based applications and web services," in *2007 IEEE International Conference on Services Computing (SCC 2007), 9-13 July 2007, Salt Lake City, Utah, USA*, 2007, pp. 68–75. [Online]. Available: https://doi.org/10.1109/SCC.2007.43

[22] M. Grechanik, D. S. Batory, and D. E. Perry, "Integrating and reusing gui-driven applications," in *Software Reuse: Methods, Techniques, and Tools, 7th International Conference, ICSR-7, Austin, TX, USA, April 15-19, 2002, Proceedings*, 2002, pp. 1–16. [Online]. Available: https://doi.org/10.1007/3-540-46020-9_1

[23] M. Grechanik, Q. Xie, and C. Fu, "Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts," in *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, 2009, pp. 9–18. [Online]. Available: https://doi.org/10.1109/ICSM.2009.5306345

[24] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.

[25] M. Grechanik, N. Prabhu, D. Graham, D. Poshyvanyk, and M. Shah, "Can software project maturity be accurately predicted using internal source code metrics?" in *Machine Learning and Data Mining in Pattern Recognition - 12th International Conference, MLDM 2016, New York, NY, USA, July 16-21, 2016, Proceedings*, 2016, pp. 774–789. [Online]. Available: https://doi.org/10.1007/978-3-319-41920-6_59

[26] Q. Xie, M. Grechanik, C. Fu, and C. M. Cumby, "Guide: A gui differentiator," in *ICSM*, 2009, pp. 395–396.

[27] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[28] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[29] P. Bille, "A survey on tree edit distance and related problems," *Theor. Comput. Sci.*, vol. 337, no. 1-3, pp. 217–239, Jun. 2005.

[30] K.-C. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, pp. 422–433, Jul. 1979.

[31] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989.

[32] P. Klein, S. Tirthapura, D. Sharvit, and B. Kimia, "A tree-edit-distance algorithm for comparing simple, closed shapes," in *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '00. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000, pp. 696–704.

[33] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An optimal decomposition algorithm for tree edit distance," *ACM Trans. Algorithms*, vol. 6, no. 1, pp. 2:1–2:19, Dec. 2009.

[34] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM TACO*, vol. 5, no. 2, pp. 10:1–10:33, Sep. 2008.

[35] M. Schwab, M. Karrenbach, and J. Claerbout, "Making scientific computations reproducible," *Computing in Science and Engineering*, vol. 2, no. 6, pp. 61–67, Nov. 2000.

[36] D. R. Slutz, "Massive stochastic testing of SQL," in *Proc. 24rd VLDB*. Morgan Kaufmann, Aug. 1998, pp. 618–622.

[37] I. Hussain, C. Csallner, M. Grechanik, C. Fu, Q. Xie, S. Park, K. Taneja, and B. M. M. Hossain, "Evaluating program analysis and testing tools with the rugrat random benchmark application generator," in *WODA*, 2012, pp. 1–6.

[38] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: achieving higher statement coverage faster," in *SIGSOFT FSE*, 2012, p. 35.

[39] I. Hussain, C. Csallner, M. Grechanik, Q. Xie, S. Park, K. Taneja, and B. M. M. Hossain, "RUGRAT: evaluating program analysis and testing tools and compilers with large generated random benchmark applications," *Softw., Pract. Exper.*, vol. 46, no. 3, pp. 405–431, 2016. [Online]. Available: https://doi.org/10.1002/spe.2290

[40] L. Alonso and R. Schott, *Random Generation of Trees: Random Generators in Computer Science*. Norwell, MA, USA: Kluwer Academic Publishers, 1995.

[41] S. Cohen and B. Kimelfeld, "Querying parse trees of stochastic context-free grammars," in *Proc. 13th ICDT*. ACM, Mar. 2010, pp. 62–75.

[42] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei, "Automating presentation changes in dynamic web applications via collaborative hybrid analysis," ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 16:1–16:11.

[43] B. A. Myers, "Why are human-computer interfaces difficult to design and implement?" Pittsburgh, PA, USA, Tech. Rep., 1993.

[44] A. M. Memon, "A comprehensive framework for testing graphical user interfaces," Ph.D. Thesis, Department of Computer Science, University of Pittsburgh, Jul. 2001.

[45] T. Ball, "On the limit of control flow analysis for regression test selection," in *Proceedings of ISSTA-98*, ser. ACM Software Engineering Notes, vol. 23,2, New York, Mar.2–5 1998, pp. 134–142.

[46] J. Bible, G. Rothermel, and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 149–183, 2001.

[47] J.-M. Kim and A. A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, 2002, pp. 119–129.

[48] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions of Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, Jul. 1993.

[49] D. Binkley, "Reducing the cost of regression testing by semantics guided test case selection," in *ICSM*, G. Caldiera and K. Bennett, Eds., Washington, Oct. 1995, pp. 251–263.

[50] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *ASE*, 2008, pp. 218–227.

[51] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On regression testing of object-oriented programs," *The Journal of Systems and Software*, vol. 32, no. 1, pp. 21–31, Jan. 1996.

[52] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *OOPSLA*, 2004, pp. 432–448.

[53] B. A. Myers, "User interface software technology," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 189–191, 1996.

[54] "Screen-scraping entry in Wikipedia," *http://en.wikipedia.org/wiki/Screen_scraping*.

[55] R. C. Miller, "End-user programming for web users." in *End User Development Workshop, Conference on Human Factors in Computer Systems*, 2003.

[56] Q. Xie and A. M. Memon, "Model-based testing of community-driven open-source GUI applications," in *ICSM*, 2006, pp. 145–154.

[57] H. Heiskanen, A. Jääskeläinen, and M. Katara, "Debug support for model-based gui testing," ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 25–34.

[58] A. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A model-to-implementation mapping tool for automated model-based gui testing," in *ICFEM*, ser. Lecture Notes in Computer Science, vol. 3785. Springer Verlag, 2005, pp. 450–464.

[59] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, and P. Jones, "Model-based testing of gui-driven applications," in *Software Technologies for Embedded and Ubiquitous Systems*, ser. Lecture Notes in Computer Science, S. Lee and P. Narasimhan, Eds. Springer Berlin Heidelberg, 2009, vol. 5860, pp. 203–214. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10265-3_19

[60] S. Roy Choudhary, M. R. Prasad, and A. Orso, "Cross-platform feature matching for web applications," ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 82–92.

[61] ——, "X-pert: Accurate identification of cross-browser issues in web applications," ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 702–711.