Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

Region-based Detection of Essential Differences in Image-based Visual Regression Testing

Haruto Tanno[1,a)] Yu Adachi[1,b)] Yu Yoshimura[1,c)] Katsuyuki Natsukawa[1,d)]
Hideya Iwasaki[2,e)]

Received: August 1, 2019, Accepted: January 16, 2020

Abstract: Visual regression testing (VRT) is a useful method for confirming that application screens are correctly displayed. VRT systems detect differences between the screens of an old version and a new version of an application to support the tester in detecting failures on the screen of the new version. One approach to VRT is image-based; i.e., before and after screenshot images are compared. It is particularly promising because screenshots are independent of the application's environment (operating system, web browser, etc.). Existing image-based VRT systems simply compare two images in pixel units and highlight pixels with differences, so if there are changes that affect the entire screen (e.g., parallel movements of screen elements), a large number of unessential differences are detected, and the essential differences are buried within them. An image-based VRT method named ReBDiff is presented that solves this problem. Before and after screen images are each divided into multiple regions, and appropriate matchings are made between corresponding regions in the two images. For each matching, differences such as shift, alteration, and addition, if any, are detected. In addition, suitable views are provided on the basis of the detected differences. By observing these views, the tester can efficiently identify the essential differences even when there are changes that affect the entire screen, e.g., parallel movements of screen elements. Experiments on a prototype system using websites for PCs and smartphones and an application screen of an Electron application demonstrated the effectiveness of the proposed method.

Keywords: image-based visual regression testing, essential differences, region pairs, difference type, pixel matching

1. Introduction

To ensure the reliability of application software, it is necessary to test the functionality of the software frequently during its life cycle, not only at the time of its initial release but also whenever new functionalities are added, the underlying operating system (OS) is updated, and so forth. The frequency of such testing is increasing due to shorter and shorter intervals between software releases. Furthermore, more and more tests are required to ensure the stability of the software due to the diversification of the system environments on which the software is deployed. For example, an environment might consist of a desktop PC, a tablet PC, and a smartphone and/or feature a software platform combining an OS and a web browser. *Regression testing* is widely used to ensure that a change made to the software does not negatively affect the system. However, the same tests must be repeated every time a new version of the software is released, and consequently the effort spent on regression testing is increasing with the scale of the target software. Automating this regression testing would thus be an effective way to save time and effort.

In the regression testing of a GUI-based application such as a web application, it is necessary to ensure that the application screens are displayed correctly. This involves two confirmations: confirming whether the application logic works correctly and the calculation results are correct and confirming whether the screen elements are laid out correctly on every application screen. The former can be automated by using a test automation tool, e.g., Selenium [*1], Appium [*2], and Sikuli [*3] [8], with suitable assertions in the scripts executed by the tool. In contrast, the latter requires that the tester carefully examines and compares the displayed layouts, which is a difficult task.

Visual regression testing [*4], or VRT for short, is a method for semi-automating the latter confirmation process. VRT detects differences between two screens of an application, typically corresponding ones before and after changes, on the basis of image information, structural information, and so forth for the screens. We call this approach to VRT in which two screenshot images are compared and only this information is used for confirmation *image-based VRT*. Since image-based VRT is applicable as long as screenshot images of application screens are available, it can be used independently of the operating environment (such as the OS

[1] NTT Laboratories, Minato, Tokyo 108–0023, Japan
[2] The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[a)] haruto.tanno.bz@hco.ntt.co.jp
[b)] yuu.adachi.gx@hco.ntt.co.jp
[c)] yuu.yoshimura.zk@hco.ntt.co.jp
[d)] katsuyuki.natsukawa.cg@hco.ntt.co.jp
[e)] iwasaki@cs.uec.ac.jp

[*1] https://docs.seleniumhq.org
[*2] http://appium.io/
[*3] https://launchpad.net/sikuli
[*4] https://github.com/mojoaxel/awesome-regression-testing

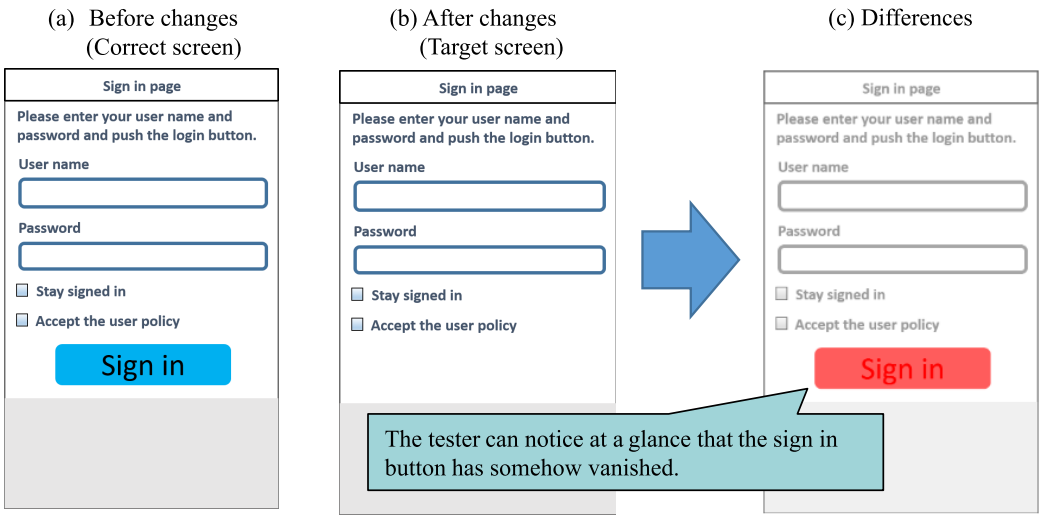


Fig. 1 Using existing image-based VRT to compare two screens in pixel units.

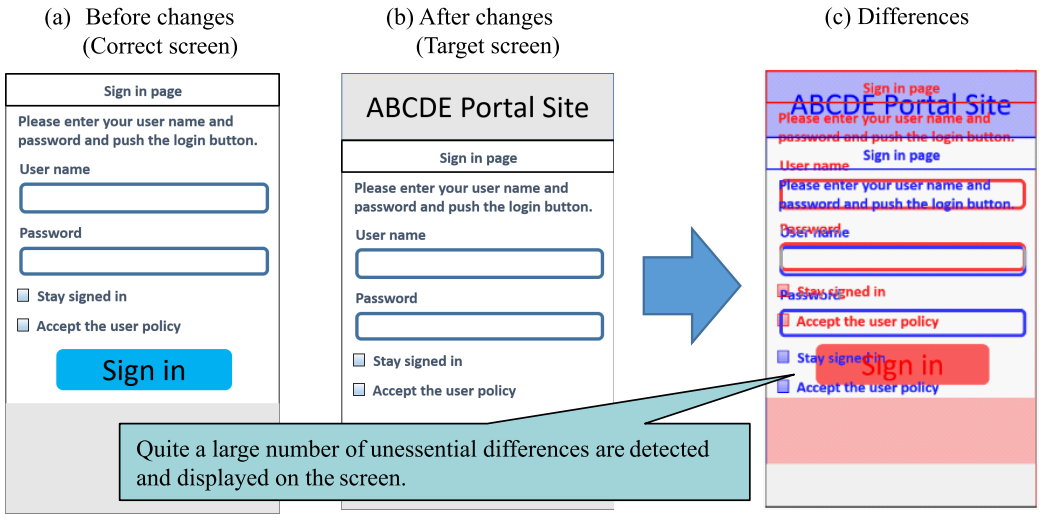


Fig. 2 Problems in comparing two screens in pixel units with existing image-based VRT.

or web browser) in which the application is executed. In addition, it can be easily used because many test automation tools provide a way to take screenshots of applications under test. Thus, there are many image-based VRT tools such as jsdiff [*5] and BlinkDiff [*6]. These tools compare two images in pixel units and highlight the ones with differences to help the tester easily and clearly identify places with differences.

As an example of applying VRT, let us consider the login screen of an authentication system for some imaginary application. **Figure 1** presents an example of differences detection by an image-based VRT system. Figures 1 (a) and (b) are screenshot images before and after changes to the application, respectively, while Fig. 1 (c) is a screenshot image in which differences are displayed. The VRT system compares the pixels in Figs. 1 (a) and (b) at the same absolute coordinate, where the origin of the coordinates is the upper-left corner. If they are the same, the system displays the pixel in grayscale in the differences image; otherwise the system displays it in red. By looking at the differences image, the tester can see at a glance that the "Sign in" button has somehow vanished in the new version of the application. In this way, image-based VRT enables the tester to recognize differences visually and, as a result, efficiently.

Unfortunately, image-based VRT systems can be problematic; if there are changes that affect the *entire* screen, it is difficult for the tester to identify the *essential differences* easily. Consider the case in which the screen design was changed by adding the header "ABCDE Portal Site" in the authentication system described above. This design change resulted in the entire (unchanged) content of the application screen being moved downward, as illustrated in **Fig. 2**. As a result, quite a large number of unessential "differences" are detected and displayed on the differences screen. The essential differences are difficult to identify because they are buried within a large number of detected differences. There are three main reasons for this problem.

- Screen elements are added or deleted in the new version in accordance with changes in functionality, screen design, and so forth, as exemplified by the case shown in Fig. 2.
- There is a region of the screen in which variable-sized elements such as advertisements and the latest news are displayed. We call such a region a *dynamic region* hereafter.

[*5] https://github.com/kpdecker/jsdiff
[*6] https://github.com/yahoo/blink-diff

- A bug in the screen element layout results in element misalignment and/or disappearance.

In these cases, it is difficult for the tester to find the essential differences by using an existing VRT system. Thus, the tester must examine the two corresponding screens carefully but is apt to overlook essential differences.

To resolve this problem, we have developed a method for making image-based VRT systems effective even in such cases. The proposed method enables the tester to compare two corresponding screens and efficiently find the essential differences.

The contributions of this paper can be summarized as follows.

- We present an image-based VRT method and its prototype system named ReBDiff (Region-based Differences detector) that enables testers to efficiently find essential differences between before and after screenshot images of an application that has been changed. It divides each image into multiple regions and makes appropriate matchings between the corresponding regions of the two images, and detects such a difference as a shift, an alteration, and an addition.
- We explain how ReBDiff can provide suitable views on the basis of the detected difference types and detailed information about them. By observing these views, the tester can find essential differences between two corresponding screens efficiently even when there are changes that affect the entire screen such as parallel movements of screen elements.
- We describe the experiments we conducted that used websites for both PCs and smartphones, and an Electron application. The results demonstrate the effectiveness of the proposed method.

Since the implementation of the proposed method is ReBDiff, we use "ReBDiff" both to indicate the method and to indicate the tool.

2. Related work

Many studies have been carried out on automating or supporting the judgment of test results [6]. This section overviews related research and tools, focusing on VRT.

2.1 Implementation-dependent VRT Systems

First we describe VRT that depends on the specific implementation technologies of the target application.

There are VRT systems that use both application screens and structural information at the same time. For web applications, the method proposed by Hori et al. [11] identified screen elements on the basis of document object model (DOM) tree information and compares two corresponding elements to determine whether the web application had been degraded. For cross-browser testing, WEBDIFF [17] and X-PERT [9] identify those places where failures have occurred by comparing the images and DOM trees of the two application screens to detect presentation failures. Ramler et al. [16] presented a method for detecting presentation failures after the user changed the magnification of the desktop in the Windows environment that utilizes both image information and screen element information.

Other approaches exploit only the structural information of application screens. The snapshot test in the JavaScript testing framework (JEST) [*7] provides a function that helps the tester ensure the absence of unexpected breakages in the UI layout on the basis of the serialized information of the displayed structures obtained in the tests. Spenkle et al. [18] presented a method for detecting differences by comparing the HTML structures of two corresponding application screens. Takahashi [19] presented a method for recording the history of API calls used for drawing an application screen and comparing the histories for two corresponding application screens. Alameer et al. [2] presented a method for detecting presentation failures after the locale of the application had been changed. A graph is constructed for each screen on the basis of the positional relationships of the elements in the corresponding DOM tree. The graphs of the correct screen and the target screen are then compared. The HTML elements with a changed appearance or a relative position are regarded as being responsible for the observed problem. Walsh et al. [22] presented a method for extracting a "responsive layout graph (RLG)" from a DOM tree and then comparing two corresponding RLGs to detect any undesired distortion of the layout in responsively designed pages.

Several methods for detecting presentation failures, which typically appear in responsively designed pages, work with only the target screen to be tested. ReDeCheck [21] detects overlapping screen elements on the basis of DOM information. VISER [4] goes even further by investigating overlapping at the pixel level, resulting in a higher precision.

These methods are useful for detecting presentation failures in regression testing, but they depend on the specific implementation technology. This means that multiple platform-dependent implementations must be prepared to enable them to be used on various platforms such as Android, iOS, and Windows.

2.2 Implementation-independent VRT Systems

Several approaches are independent of the implementation technology. They attempt to identify the problem from only the application screen images.

The jsdiff and BlinkDiff image-based VRT tools detect differences between two images in pixel units. Similar approaches were taken in VISOR [12] and by Mahajan and Halfond [14]. These tools and methods focus mainly on comparing old and new versions of an application in regression testing. They are effective when the two images to be compared are almost the same, with only minor differences, as exemplified in Fig. 1. They are not effective when there are changes that affect the entire screen, as exemplified in Fig. 2. Mahajan and Halfond [15] adjusted their method to absorb small differences at the pixel level, but it is still not effective when the positional shift is more than negligible. Lin et al. [13] presented a method that calculates the similarity between the correct screen and the target screen by using several indices such as a histogram of similarity. If the similarity falls below a certain threshold, the target screen under test is presumed to have problems. Although this method can be used to roughly estimate the similarity between two screens on Android terminals with different resolutions, it cannot localize the differences.

[*7] https://jestjs.io/

Other approaches do not depend on image-based VRT. Visual GUI testing tools [3] such as Sikuli [8] make use of image recognition techniques. Since they treat matching objects on an application screen as images, they can only be used as long as the target objects are present on the screen. They are aimed at ensuring that images are displayed on the screen as expected; they cannot ensure that the screen elements are properly placed without distortion of the screen's appearance. Bajammal and Mesbah [5] presented a method that analyzed a screenshot image of canvas elements in HTML5, identified every visual object and its attributes, and constructed a layered structure of the visual objects for use in generating suitable assertions for the image. Assertions generated for the image of the correct screen can be applied to the image of the target screen to be tested. Unfortunately, their method is applicable only to the canvas in HTML5.

Image-based VRT is flexible and offers two advantages in particular.

- It can be used as long as screenshot images of the application screens are available. Thus, it does not depend on specific implementation technologies such as the OS and web browser.
- It can be easily combined with test automation tools for practical application because such tools generally provide a functionality for obtaining screenshot images.

We have developed a method for making image-based VRT applicable to situations in which there are changes that affect the entire screen and have therefore expanded the scope of its potential application.

3. Proposed method

3.1 Scope and Requirements

To summarize the discussion in Section 2, there are two use cases for VRT. One use case is comparing two corresponding screen images of the old and new versions of an application in the same environment. For example, the "Sign in" screen of the old version is compared with that of the new version in the Chrome browser. The other use case is testing the same version of the target application in various environments, in which a screen image for one environment is compared with the corresponding one for another environment. This includes cross-browser testing and testing on various Android devices.

The application scope of ReBDiff is the first use case with the aim of applying image-based VRT techniques to regression testing even when there are changes that affect the entire screen, exemplified by the three cases described in Section 1. The second use case is outside the application scope of ReBDiff. Please note that our aim is to detect differences between two given images, not to determine whether each detected difference is a bug. We assume that the tester is responsible for making that determination.

The application scope defined for ReBDiff means that there are three requirements for a system that supports the tester in detecting and checking essential differences between two application screens by using image-based VRT.

Requirement 1 Each difference can be detected at a level of granularity that makes it easy for the tester to identify the difference.

Requirement 2 All the regions shifted due to changes that affected the entire screen can be checked together.

Requirement 3 Detected differences are displayed with good visibility.

3.2 Features of ReBDiff

ReBDiff has three features in particular.

First, it detects essential differences in two stages. In the first stage, it roughly detects differences between two corresponding *regions*, one in the correct screen and the other in the target screen under test, and labels each detected difference with one or two *difference type*s among Shift, Addition, Deletion, Alteration, and Scaling. Here, a region is a rectangular section in an image in an application screen. In the second stage, ReBDiff applies an existing image-based VRT method to a pair of corresponding regions. This two-stage process enables the tester to check the differences roughly at the region level (Requirement 1). In addition, the tester can identify parallelly moved regions easily in the first stage (Requirement 2).

Second, ReBDiff groups together regions labeled Shift that have the same direction and amount of movement. Thus, the tester can check these regions together, not one by one (Requirement 2). This feature contributes not only to making detected differences visible (Requirement 3) but also to reducing the burden on the tester.

Third, ReBDiff highlights each difference in accordance with its type. Thus, the tester can recognize detected differences with good visibility (Requirement 3).

Figure 3 shows an overview of ReBDiff. Given two images, one of the correct screen and one of the target screen, ReBDiff displays two views; one for differences at the region level and the other for differences between corresponding regions in pixel units.

3.3 Difference Types

ReBDiff divides the correct and target screens to be tested into regions and detects differences, as presented in **Fig. 4**. For every detected difference, ReBDiff assigns one or two difference types. Currently there are five difference types.

Shift There are highly similar regions in the correct and target screen images, but their positions differ.

Addition The target screen image has a region with no corresponding region in the correct screen image.

Deletion The correct screen image has a region with no corresponding region in the target screen image.

Alteration There are similar regions in the correct and target screen images, but their similarity is not high.

Scaling There are similar regions in the correct and target screen images, but their sizes differ.

Types Shift, Scaling, and Alteration are cases in which there are very similar but not the same regions in both screens. Types Addition and Deletion are exemplified by Region 4' and Region 2 in Fig. 4, respectively.

These five difference types should suffice for the following reason. Differences can be divided into two groups: 1) those between

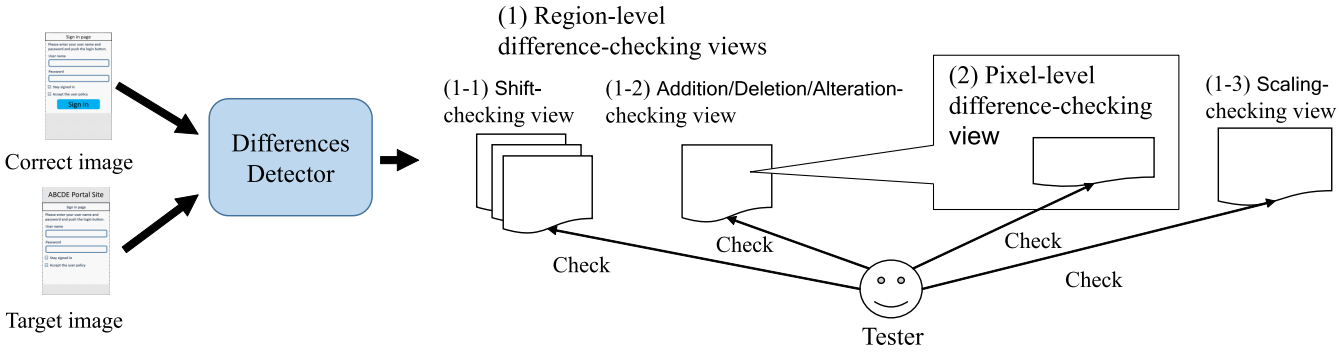


Fig. 3 Overview of ReBDiff.

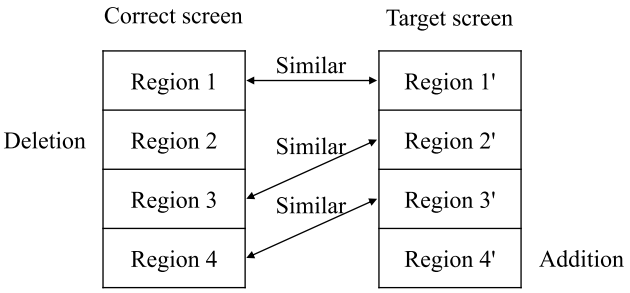


Fig. 4 Detecting differences in region pairs.

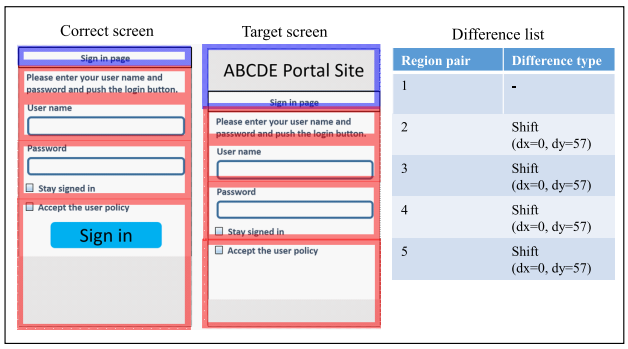


Fig. 5 Shift-checking view (group identifier is 1).

corresponding regions in the correct and target screen images and 2) those without corresponding regions in the two screen images. The former can be further classified into Scaling and Alteration on the basis of the similarity level. In addition, if the positions of corresponding regions are not the same, Shift is added. For the latter group, the differences can be further classified into Addition in which a new region is added to the target screen, and Deletion in which a region is deleted from the correct screen.

Precisely speaking, a difference type is assigned to a region pair explained in Section 4. For the example shown in Fig. 4, region pairs $(null, 4')$ and $(2, null)$, where *null* means that there is no corresponding region, are associated with Addition and Deletion, respectively. ReBDiff regards regions with high similarity, i.e., greater than a predefined threshold, and with a sufficiently small size difference, i.e., within a predefined threshold, as the "same" and does not detect them as a difference.

The similarity of two regions is calculated on the basis of whether the larger region in height includes an area similar to the smaller region. This is described in more detail in Section 4. Thus, there may be cases where similarity is high for regions with different heights. Since such regions need to be checked by the tester, Scaling is added for the regions (a region pair) as a difference type.

Among the five types, Shift, Scaling, and Alteration have additional information on the difference. This information is made explicit by using the following notations.

- Shift (dx, dy): dx and dy are the amounts of movement in the horizontal and vertical directions, respectively.
- Scaling (sx, sy): sx and sy are the scaling factors in the horizontal and vertical directions, respectively.
- Alteration d: d represents detailed information on differences at the pixel level.

Please note that two types, namely "Alteration and Shift" or "Scaling and Shift," might be assigned to a detected difference. The details are described in Section 4.3.

3.4 Difference Checking by Tester

First, the tester checks the Shift differences (Fig. 3 (1-1)). The tester can check each group of region pairs with the same direction and amount of movement as a whole by using the *Shift-checking view*. For example, **Fig. 5** presents an instance of this view when the input screens are those in Figs. 2 (1) and (2). In this example, both the correct and target screens are divided into five regions. Since four region pairs except the upper-most one move together vertically a distance of 57 pixels, they are displayed within a red frame. Seeing this view, the tester judges that this difference is not a problem because the movement of these four region pairs is the result of the change in the upper-most part of the screen. In this example, there is only a single group of region pairs. When there are multiple groups, the tester checks them one by one by using the Shift-checking view.

After checking the Shift differences, the tester proceeds to check Addition, Deletion, and Alteration differences in the *Addition/Deletion/Alteration-checking view* (Fig. 3 (1-2)). For the example in Fig. 2, **Fig. 6** presents the view used for this check, where two differences in Region pairs 1 and 5 are detected. The tester is shown the type of each difference at the right side of the view. When the tester selects a difference in the list (the selected difference is displayed in yellow), the corresponding region is highlighted.

The tester can see the details for an Alteration difference in two ways.

- The tester can compare two regions in pixel units by investi-

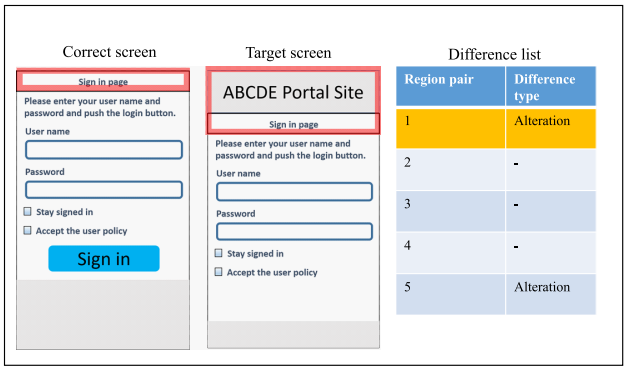


Fig. 6 Addition/Deletion/Alteration-checking view.

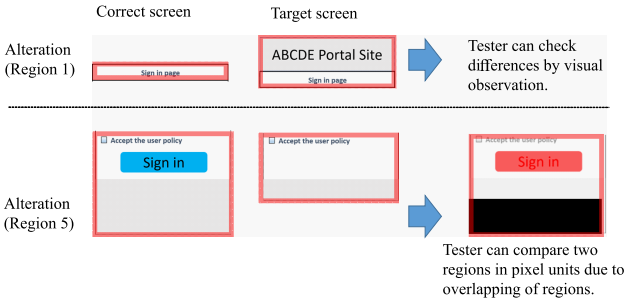


Fig. 7 Checking for Alteration differences.

gating their overlapped image generated by ReBDiff. When the differences are local and not so large, the place where changes occur can be easily recognized.

- The tester can check the differences by visual examination. Although a visual examination is burdensome, ReBDiff reduces the burden because the area of the region to examine is (much) smaller than the entire screen.

For the example in **Fig. 7**, region pair 5, where the "Sign in" button has vanished in the test screen, can be checked by examining the overlapped image. In contrast, region pair 1's change is difficult to grasp by examining the overlapped image, so a visual examination is needed.

Finally the tester checks for Scaling differences by using the *Scaling-checking view* (Fig. 3 (1-3)). For each Scaling difference, scaling factors (both horizontal and vertical) in percentage are displayed in the list at the right side of the view.

In this way, comparing differences at the region level makes it possible to roughly check the differences in accordance with the type(s) of each difference.

4. Differences Detector

We implemented our differences detector, shown in Fig. 3, by utilizing computer vision techniques [10]. Specifically, we used the Python bindings of OpenCV 3.1.0.

Let C be the image of the correct screen and T be the image of the target screen under test. Differences between C and T are detected automatically in three steps.

Step 1 ReBDiff divides C into m regions $c_1, \ldots, c_m$ and T into n regions $t_1, \ldots, t_n$.

Step 2 ReBDiff extracts *region pairs*, each of which consists of a region in C and a region in T that are similar to each other, and creates a list PL of region pairs.

Step 3 ReBDiff assigns one or two difference types to every pair in PL.

Let r, r_1, and r_2 be regions. We assume that $ul(r)$ and $lr(r)$ represent the coordinates of the upper-left corner of r and that of the lower-right corner of r, respectively. In addition, $wd(r)$, $ht(r)$, and $size(r)$ represent the width, height, and size (number of pixels) of r, respectively. We also assume that $sim(r_1, r_2)$ is the similarity of r_1 and r_2. We calculate $sim(r_1, r_2)$ by using the `cv2.matchTemplate` method, which performs template matching, and the `cv2.minMaxLoc` method, which obtains the maximum value of the similarity from the template matching results.

Hereafter, we will explain each step.

4.1 Step 1: Divide Images into Regions

The first step is to divide C and T into regions in accordance with sub-steps 1-1 to 1-4 below. There are two modes for dividing an image into regions. One is *H-mode*, which divides an image horizontally. This mode is used for vertical screen images in which their contents are horizontally arranged such as web pages for mobile devices and screens of Android/iOS applications. The other is *HV-mode*, which divides an image horizontally first and then further divides each divided region vertically. This mode is used for screen images in which the contents are both vertically and horizontally arranged such as web pages for PCs and Windows native applications. The tester can specify which mode to use in accordance with the features of the screen under test.

The procedure for dividing an image into regions comprises four steps.

Sub-step 1-1 ReBDiff applies Canny edge detection [7] to C to detect the edges. Specifically, by using the `cv2.Canny` method, ReBDiff generates a binary image in which pixels representing the edges are white and the other pixels are black. Then, ReBDiff performs line detection in the horizontal direction on C as follows. If there is a row where the number of white pixels, which represent edges, exceeds $wd(C) \times S_L$ in the generated binary image, ReBDiff regards the row as a line. If multiple consecutive rows are regarded as lines, only the middle one is taken and the others are deleted. S_L is a predefined parameter with a value that should be empirically determined so that an appropriate division of images into regions can be obtained. In our experiments, described in Section 4, we set S_L to 0.8. Line detection is used to divide C into K_1 regions: $C = r_1^1, \ldots, r_{K_1}^1$.

Sub-step 1-2 Starting from r_1^1, ReBDiff repeatedly concatenates adjacent regions until the area of the concatenated region exceeds a predefined threshold. Let the concatenated region be r_1^2. Then ReBDiff performs the same process starting from the region in which the concatenation had terminated. Repetition of this process until no region remains results in C having K_2 regions: $C = r_1^2, \ldots, r_{K_2}^2$. The threshold is $wd(C) \times S_R$, where S_R is a predefined parameter. The following is the pseudo code for this process.

```
limit ← wd(C) * S_R;
h ← 0; from ← 1; j ← 1;
for i ← 1 to K_1 do begin
  h ← h + ht(r_i^1);
```

if $h \geq limit$ **then begin**
$r_j^2 \leftarrow$ a region where r_{from}^1 to r_i^1 are combined;
$h \leftarrow 0$; $from \leftarrow i+1$; $j \leftarrow j+1$;
end;
end;
if $from \leq K_1$ **then begin**
$rest \leftarrow$ a region where r_{from}^1 to $r_{K_1}^1$ are combined;
$r_{j-1}^2 \leftarrow$ a region where r_{j-1}^2 and $rest$ are combined;
end;
$K_2 \leftarrow j-1$;

If the value of S_R is too small, many small regions are generated. As a result, it would take much time to calculate the region pairs in Step 2. In addition, the correspondence accuracy for region pairs would be reduced. Therefore, it is necessary to set an appropriate value of S_R empirically to generate moderately sized regions. In our experiments, we set S_R to 0.1.

Sub-step 1-3 From $r_1^2, \ldots, r_{K_2}^2$, ReBDiff creates $C = r_1^3, \ldots, r_{K_3}^3$ by merging a single-colored region and an adjacent multi-colored region into a single region. This step is necessary because if many single-colored regions are eventually generated in Step 1, it is likely that Step 2 cannot properly associate a region in the correct image with a region in the target image under test. As a result of this merging, C does not contain a single-colored region if the processing image is not single-colored. Pseudo code for this process is as follows.

$from \leftarrow 1$; $j \leftarrow 1$;
for $i \leftarrow 1$ **to** K_2 **do begin**
$rr \leftarrow$ a region where r_{from}^2 to r_i^2 are combined;
if rr is a multi-colored region
$r_j^3 \leftarrow rr$;
$from \leftarrow i+1$; $j \leftarrow j+1$;
end;
end;
if $from \leq K_2$ **then begin**
$rest \leftarrow$ a region where r_{from}^2 to $r_{K_2}^2$ are combined;
$r_{j-1}^3 \leftarrow$ a region where r_{j-1}^3 and $rest$ are combined;
end;
$K_3 \leftarrow j-1$;

Sub-step 1-4 For H-mode, $r_1^3, \ldots, r_{K_3}^3$ is directly the resulting list of regions. For HV-mode, ReBDiff performs the same process (Sub-steps 1-1 to 1-3) for every r_i^3 ($1 \leq i \leq K_3$), where line detection in Step 1-1 is done in the vertical direction.

Following the above steps, ReBDiff obtains a list of regions for C, i.e., $c_1, \ldots, c_m$. ReBDiff performs a similar process for T and obtains $t_1, \ldots, t_n$.

4.2 Step 2: Generating List of Region Pairs

The next step is to create a list of region pairs by repeatedly associating a region in C with a region in T one by one on the basis of the similarity between the two regions. Since the cost of calculating similarities for all possible pairs is too large, ReBDiff calculates the similarity only for those regions with coordinates close to each other.

Let R_H and R_V be ranges in the horizontal and vertical directions, respectively, used to search the corresponding region, and let S_P be the similarity threshold used to judge whether two regions are pairable. As before, PL is a list of region pairs, which is initially empty.

We define the similarity between two regions as the result of template matching [10] on one region using the other region as a template image. Of the two regions, the one with the smaller area is used as the template image. PL is created using a two-step process.

Step 2-1 For region c in C, ReBDiff selects regions t from T, each of which satisfies three conditions: (1) t exists in a rectangular region for which the upper-left coordinate is $ul(c) - (R_H, R_V)$ and the lower-right coordinate is $lr(c) + (R_H, R_V)$; (2) $wd(c) = wd(t)$; and (3) $sim(c,t) > S_p$. Let t be the region with the highest value of similarity with c among the regions selected from T. If such a t exists, ReBDiff adds a region pair (c,t) to PL and removes t from T; otherwise, ReBDiff adds a region pair $(c, null)$ to PL. This procedure is performed in order from c_1 to c_m.

Step 2-2 For every region t in T that was not selected in Step 2-1, ReBDiff adds a region pair $(null, t)$ to PL.

If the value of S_P is too small, many inappropriate region pairs that do not contain *null* may be created. On the other hand, if the threshold is too large, two regions that should be paired and labeled Scaling or Alteration may not be paired, resulting in many $(c, null)$ and $(null, t)$ region pairs (Deletion and Addition) being generated. Because the former situation is more serious, it is necessary to empirically determine that S_P is sufficiently large, but not too large. In our experiments, we set S_P to 0.5.

4.3 Step 3: Assigning Difference Types to Region Pairs

The final step is to assign one or two appropriate difference types to every region pair in PL. Let S_M ($S_M > S_P$) be the threshold for similarity. The following logic is used to assign difference type(s) to every region pair $p = (c,t)$ in PL, where $attach(p, ty)$ assigns ty to p.

if $c = null$ **then begin** $attach(p,$ Addition$)$; **return end**
else if $t = null$ **then begin** $attach(p,$ Deletion$)$; **return end**;
if $ul(c) \neq ul(t)$ **then begin**
$(dx, dy) \leftarrow ul(t) - ul(c)$;
$attach(p,$ Shift $(dx, dy))$ **end**;
if $sim(c,t) < S_M$ **then begin**
$d \leftarrow$ differences between c and t in pixel units;
$attach(p,$ Alteration $d)$ **end**;
else if $size(c) \neq size(t)$ **then begin**
$(sx, sy) \leftarrow (wd(t)/wd(c),\ ht(t)/ht(c))$;
$attach(p,$ Scaling $(sx, sy))$ **end**
return;

Please note that both Shift and Scaling or both Shift and Alteration might be assigned to a pair. In fact, changes that can be regarded as a parallel movement and also regarded as an alteration or a scaling are commonly seen. Even for such cases, the tester can check each difference type assigned to the pair by using the checking view corresponding to the type, as described in Section 3.4.

Table 1 Target screens.

Experiment	Target screen	App. type	Screen size	No. of type 1 change(s) and description	No. of type 2 change(s)
1	Am	Mobile	411 × 1327	1: Change in design in upper side	1
	Ap	PC	839 × 928	1: Change in design in upper side	1
	Bm	Mobile	411 × 2061	1: Deletion of logo at top	1
	Bp	PC	1042 × 1813	1: Deletion of logo at top	1
	Cm	Mobile	411 × 5672	1: Deletion of advertisement at top	1
	Cm11	Mobile	411 × 5672	1: Deletion of advertisement at top	10
	Dp	PC	1097 × 4200	1: Deletion of advertisement in upper side	1
	Dp11	PC	1097 × 4200	1: Deletion of advertisement in upper side	10
	Em	Mobile	411 × 4490	2: Deletion of advertisement in upper side and link button in middle of screen	2
2	Xe	Electron	765 × 593	2: Change in UI in upper and bottom side	0

Finally, ReBDiff assigns the same group identifier to all pairs in PL with Shift that have the same movement amounts, i.e., the same (dx, dy) value.

5. Experiments

5.1 Research Questions

To evaluate the effectiveness of ReBDiff, we conducted experiments to answer two research questions.

RQ1 Can ReBDiff detect all differences between the correct screen and the target screen under test? Is the number of detected differences as small as possible? Is the tester's effort for confirming the detected differences sufficiently small?

RQ2 What are the differences in the discovery rate and confirmation time compared with those for manual confirmation?

5.2 Method

To answer RQ1, we conducted two experiments.

The first one used target screens with embedded artificial mutations representing changes. We prepared images of correct screens by taking screenshots of real-world applications. We then created target images by embedding changes (described below) into the correct images.

- Additions, deletions, shifts, and scalings of screen elements. In some cases of shifting, two screen elements became overlapped.
- Slight alterations of screen elements.
- Changes in line feed positions and fonts.

The second experiment used screens in which there were *actual* changes in a real-world application. We prepared screenshot images of corresponding old version and new version screens.

To answer RQ2, we asked four participants to detect differences in two ways, i.e., by visually checking the entire image manually and by using ReBDiff. We then measured the rate of differences discovery and the time required for confirmation. For each manual detection, we prepared an Excel sheet with the correct image and the target image side by side so that the participants were able to perform visual confirmation as efficiently as possible not only by looking at the display but also by referring to the Excel sheet. For each participant, the target screen for ReBDiff confirmation differed from that for manual confirmation to prevent learning effects.

The parameters and thresholds at each step in the differences detection were adjusted by using data from real-world websites for PCs and smartphones (excluding websites related to the target screens used in this experiment) so that differences were properly detected. The parameter and threshold values were $S_L = 0.8$, $S_R = 0.1$, $S_P = 0.5$, and $S_M = 0.97$.

5.3 Target Screens

Table 1 lists the target screens used in the experiments. Am and Ap are login screens for the Japan Pension Service Nenkin net (mobile and PC versions, respectively). Bm and Bp are login screens for the Internet banking service of the Japan Post Bank (mobile and PC versions, respectively). Cm and Cm11 are the top pages of NTT East's mobile web service, Dp and Dp11 are the top pages of NTT DOCOMO's "My docomo" service for PC web, and Em is the top page of NTT West's mobile web service. Am and Ap have a rather simple design and a small screen. Dp, Dp11, and Em have more complicated designs and larger screens.

For each target screen, the target image had

- changes that affected the entire screen (*type 1* changes) such as insertion of a logo at the top of the page, and
- changes that did not affect the entire screen (*type 2* changes) such as deletion of the login button.

For Cm11 and Dp11, type 2 changes were embedded as much as possible in the entire screen. There were ten such changes.

Xe is the screen of an Electron application (2 KL), which had been developed at NTT. Its target image had two type 1 changes and no type 2 changes. Since the Xe screen was modified by the addition of a new function to the application, it was necessary to confirm that unexpected changes on the new version's screen did not occur elsewhere.

For all target screen displays, we observed that the problem shown in Fig. 2 occurred when using an existing image-based VRT that performed difference detection in pixel units.

We obtained screenshot images of PC and mobile web screens on the Chrome browser in Windows 10 by using Full Page Screen Capture, which is a Chrome extension. For the mobile web pages, we used Chrome's developer tool to display the pages with the screen size of the mobile version (we used the size of the Pixel2 XL terminal) and then captured screenshot images. For the Electron application, we obtained screenshots by pushing the Alt and PrintScreen keys, the traditional way to get screenshots in Windows.

In addition, for Cm11 and Dp11, we asked the participants to detect differences in two ways: by using ReBDiff and by manual checking of the entire image.

5.4 Results

5.4.1 RQ1

Table 2 presents the results of applying ReBDiff to each target screen: the number of detected differences for each difference

Table 2 Results for RQ1.

Target screen	No. of detected differences					Ratio of area					Difference detection rate
	Shift	Addition	Deletion	Alteration	Scaling	Shift	Addition Deletion	Alteration (confirmable in pixel units)	Alteration (not confirmable in pixel units)	Scaling	
Am	2	1	1	1	0	90.3%	5.8%	11.7%	0%	0%	2/2
Ap	1	0	0	2	0	62.7%	0%	38.6%	23.7%	0%	2/2
Bm	1	2	2	0	0	93.2%	6.8%	0%	0%	0%	2/2
Bp	1	0	0	1	1	87.6%	0%	76.4%	0%	12.4%	2/2
Cm	1	0	1	1	0	99.6%	0.4%	2.3%	0%	0%	2/2
Cm11	2	3	4	7	0	96.4%	3.6%	13.4%	0%	0%	11/11
Dp	1	1	1	1	0	92.9%	4.3%	2.8%	0%	0%	2/2
Dp11	1	1	1	10	0	93.0%	4.3%	23.1%	0%	0%	11/11
Em	2	0	0	4	0	92.2%	0%	13.1%	0%	0%	4/4
Xe	1	0	0	2	0	68.2%	0%	0%	56.6%	0%	2/2

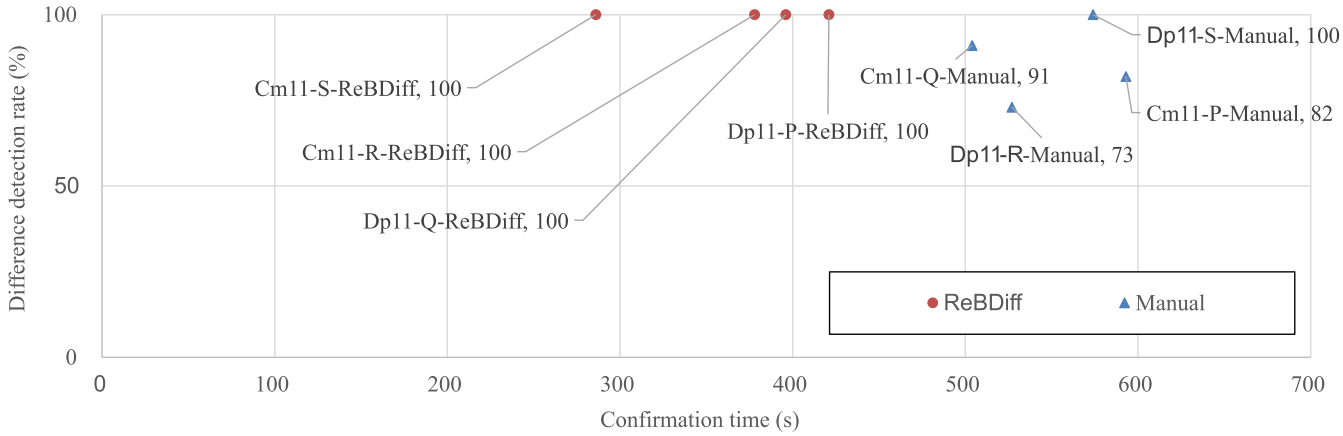


Fig. 8 Difference detection rates and confirmation times for RQ2.

type and the ratio of the area of the regions in which differences were detected to the total area. Here, the total area is the sum of the area of the correct image and that of the target image under test, which can be regarded as the entire area to be checked in order to detect differences. Since manual checking without ReBDiff requires that the total area be entirely checked, the smaller the area of the regions for which ReBDiff will be used to detect differences, the greater the effectiveness of ReBDiff.

For all target screens, ReBDiff detected all (type 1 and type 2) changes. Though the ratios of the area of detected Shift were large, i.e. over 90% for seven cases and at least 62.7%, the number of detected Shift differences in each case was 1 or 2. Therefore, these differences should be confirmable without much effort by the tester.

Examining the Addition and Deletion differences in detail, we see that two regions that should have been paired to form a region pair and detected as a single Alteration difference were detected as an Addition difference and a Deletion difference. This was because they were not similar enough to be paired. In such cases, the tester must expend much effort in checking them because the contents of both the Addition region and the Deletion region must be visually checked to identify the differences between them. However, the ratio of each area for the Addition and Deletion differences was less than 6.8% of the total area, which is not particularly large.

The Alteration differences that were confirmable by comparison in pixel units would not impose a large burden on the tester even though the ratio of the area was as high as 76.4%. This is because ReBDiff provides an overlapping view (Fig. 7) that helps the tester compare the two regions.

The Alteration differences for Ap and Xe were impossible to confirm by comparison in pixel units. Though the tester would have to visually check these differences, the burden would be greatly reduced by using ReBDiff because ReBDiff narrowed down the region to be checked (to 23.7% for Ap and 56.6% for Xe). This means that the tester would not need to visually check the entire screen.

5.4.2 RQ2

The difference discovery rate and time required for confirmation are plotted in **Fig. 8** for both using ReBDiff and manual confirmation. In both cases, there were (the same) four participants, referred to here as P, Q, R, and S. The target screens were Cm11 and Dp11, both of which had one type 1 change and ten type 2 changes. A plotted point labeled "*X*-*Y*-ReBDiff" indicates that participant *Y* confirmed target screen *X* by using ReBDiff and a plotted point labeled "*X*-*Y*-Manual" indicates that participant *Y* confirmed target screen *X* manually. When ReBDiff was used, all differences were detected by all participants, whereas when manual confirmation was used, some differences were overlooked. In addition, the times required for confirmation using ReBDiff were much shorter than those using manual confirmation. These results show that ReBDiff improves the difference detection rate and reduces the confirmation time.

Six differences were overlooked in the manual confirmation.

- Differences in font type: 2 instances
- Difference in font size: 1 instance
- Difference in sentence content: 1 instance
- Change in icon position: 1 instance

- Change in logo position: 1 instance

Two participants failed to manually detect differences in font type. Though such differences are difficult to detect visually, they are easily found by ReBDiff by using overlapping and comparing the two regions in pixel units. Some participant also overlooked differences that were relatively easy to find such as a difference in font size and changes in an icon's/logo's position. For such cases, by using ReBDiff, the participants had only to check a limited part of the entire screen and were able to compare an Alteration difference in pixel units. As a result, they found the differences. This demonstrates the effectiveness of using ReBDiff.

5.5 Discussion

In the experiments, ReBDiff was able to detect all differences as shown Table 2. However, in some cases ReBDiff may overlook a difference. For example, if the area of changes in a region in the target screen is very small relative to the area of the region, this difference might not be detected. Adjusting the threshold values (S_P and S_M) enables the alteration of regions to be judged more strictly, which reduces such missed detections. Such stricter judgment of equivalences can cause many unessential differences. There is thus a trade-off between reducing the number of oversights and reducing the number of detected unessential differences. An appropriate policy for this might be, in tests where even a few oversights are unacceptable, the equivalence of the two sections should be strictly determined. In tests where time is a priority and a few oversights are acceptable, the equivalence of the two sections should be determined less strictly.

In the experiments, we used screens for PC web and mobile web services and an Electron application with different display sizes and different implementation technologies. The embedded change patterns were created on the basis of interviews with developers who had been mainly developing mobile web applications. Since the effectiveness of the proposed method was demonstrated using these screens and change patterns, the proposed method should be widely applicable. To verify that it can be applied more widely and more generally, it needs to be evaluated using a wide variety of application types, e.g., Android and iOS native applications. Future work also includes interviewing testers at a wider variety of development sites.

As described in Section 1, there may be dynamic regions such as those for advertisements and news articles within a screen, and these regions are detected as differences. If many such differences are detected in practical use of ReBDiff the time for checking differences will be longer. A promising method for overcoming this problem is to specify a mask area on the basis of the relative positional relationships of multiple screen elements and use it to remove the dynamic regions from the screen [1].

If the correct screen and the target screen greatly differ, two problems may occur.

- The coordinates of the corresponding regions in the two screens would be too far apart. As a result, a large number of Addition and Deletion differences would be detected because ReBDiff would be unable to create region pairs properly.
- ReBDiff would create an incorrect region pair and detect it as an Alteration difference. If many such Alteration differences are detected, the tester would probably get confused.

A practical solution when more than a certain number of differences are detected for the target screen is to carry out a visual confirmation without using ReBDiff's checking views.

6. Conclusion

Our proposed image-based visual regression testing system, ReBDiff, divides each of the images of the two application screens to be compared into multiple regions, makes appropriate matchings between corresponding regions in the two images, and detects differences on the basis of the matchings. By using ReBDiff, the tester can identify essential differences between the two screens efficiently even when there are changes that affect the entire screen, e.g., parallel movements of screen elements. Experiments using screens for PC web and mobile web services and an Electron application demonstrated the effectiveness of the proposed method.

A product [20] incorporating the technology used in ReBDiff is currently being used at NTT group companies. Future work includes improving ReBDiff by reflecting the feedback and comments of actual users.

References

[1] Adachi, Y., Tanno, H. and Yoshimura, Y.: Masking Dynamic Content Areas Based on Positional Relationship of Screen Elements for Visual Regression Testing (in Japanese), *JSSST Computer Software*, Vol.36, No.4, pp.53–59 (2019).

[2] Alameer, A., Mahajan, S. and Halfond, W.G.J.: Detecting and Localizing Internationalization Presentation Failures in Web Applications, *2016 IEEE International Conference on Software Testing, Verification and Validation*, *ICST 2016*, pp.202–212 (2016).

[3] Alegroth, E., Feldt, R. and Ryrholm, L.: Visual GUI Testing in Practice: Challenges, Problems and Limitations, *Journal of Empirical Software Engineering*, Vol.20, No.3, pp.694–744 (2015).

[4] Althomali, I., Kapfhammer, G.M. and McMinn, P.: Automatic visual verification of layout failures in responsively designed web pages, *12th IEEE Conference on Software Testing*, *Validation and Verification*, *ICST 2019*, pp.183–193 (2019).

[5] Bajammal, M. and Mesbah, A.: Web Canvas Testing Through Visual Inference, *11th IEEE International Conference on Software Testing*, *Verification and Validation*, *ICST 2018*, pp.193–203 (2018).

[6] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S.: The Oracle Problem in Software Testing: A Survey, *IEEE Trans. Software Engineering*, Vol.41, No.5, pp.507–525 (2015).

[7] Canny, J.: A Computational Approach to Edge Detection, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol.8, No.6, pp.679–698 (1986).

[8] Chang, T., Yeh, T. and Miller, R.C.: GUI Testing Using Computer Vision, *28th International Conference on Human Factors in Computing Systems*, *SIGCHI 2010*, pp.1535–1544 (2010).

[9] Choudhary, S.R., Prasad, M.R. and Orso, A.: X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications, *35th International Conference on Software Engineering*, *ICSE 2013*, pp.702–711 (2013).

[10] Kaehler, A. and Bradski, G.: *Learning OpenCV: Computer Vision with the OpenCV Library*, O'Reilly Media (Aug. 2009).

[11] Hori, A., Takada, S., Tanno, H. and Oinuma, M.: An Oracle based on Image Comparison for Regression Testing of Web Applications, *27th International Conference on Software Engineering and Knowledge Engineering*, *SEKE 2015*, pp.639–645 (2015).

[12] Kıraç, F., Aktemur, B. and Sözer, H.: VISOR: A Fast Image Processing Pipeline with Scaling and Translation Invariance for Test Oracle Automation of Visual Output Systems, *Journal of Systems and Software*, Vol.136, pp.266–277 (2017).

[13] Lin, Y., Rojas, J.F., Chu, E.T. and Lai, Y.: On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices, *IEEE Trans. Software Engineering*, Vol.40, No.10, pp.957–970 (2014).

[14] Mahajan, S. and Halfond, W.G.J.: Finding HTML Presentation Failures Using Image Comparison Techniques, *29th ACM/IEEE International Conference on Automated Software Engineering*, *ASE 2014*,

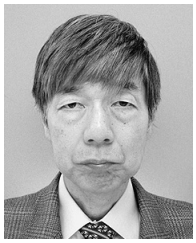
pp.91–96 (2014).
[15] Mahajan, S. and Halfond, W.G.J.: Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques, *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015*, pp.1–10 (2015).
[16] Ramler, R., Wetzlmaier, T. and Hoschek, R.: GUI Scalability Issues of Windows Desktop Applications and How to Find Them, *Companion Proc. ISSTA/ECOOP 2018 Workshops*, pp.63–67 (2018).
[17] Roy Choudhary, S., Versee, H. and Orso, A.: Webdiff: Automated identification of cross-browser issues in web applications, *2010 IEEE International Conference on Software Maintenance, ICSM 2010*, pp.1–10 (2010).
[18] Sprenkle, S., Pollock, L., Esquivel, H., Hazelwood, B. and Ecott, S.: Automated Oracle Comparators for TestingWeb Application, *18th IEEE International Symposium on Software Reliability, ISSRE 2017*, pp.117–126 (2007).
[19] Takahashi, J.: An Automated Oracle for Verifying GUI Objects, *ACM SIGSOFT Software Engineering Notes*, Vol.26, No.4, pp.83–88 (2001).
[20] Tanno, H. and Adachi, Y.: Support for Finding Presentation Failures by Using Computer Vision Techniques, *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2018*, pp.356–363 (2018).
[21] Walsh, T.A., Kapfhammer, G.M. and McMinn, P.: ReDeCheck: An Automatic Layout Failure Checking Tool for Responsively Designed Web Pages, *26th International Symposium on Software Testing and Analysis, ISSTA 2017*, pp.360–363 (2017).
[22] Walsh, T.A., McMinn, P. and Kapfhammer, G.M.: Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pp.709–714 (2015).

Haruto Tanno is currently a researcher in the Software Innovation Center at Nippon Telegraph and Telephone Corporation (NTT), Tokyo, Japan. He received B.E. and M.E. degrees from The University of Electro-Communications, in 2007 and 2009. He joined NTT in 2009. His research interests include software testing and debugging. He is a member of the IPSJ.

Yu Adachi is currently a researcher in the Software Innovation Center at Nippon Telegraph and Telephone Corporation (NTT), Tokyo, Japan. He received B.E. and M.E. degrees from The University of Electro-Communications, Tokyo, in 2007 and 2009. He joined NTT in 2009. His current research area is software engineering.

Yu Yoshimura is currently a researcher in the Software Innovation Center at Nippon Telegraph and Telephone Corporation (NTT), Tokyo, Japan. He received B.E. and M.E. degrees from the Tokyo University of Science in 2012 and 2014. He joined NTT COMWARE Corporation, Tokyo, in 2014. His current research interests include software engineering.

Katsuyuki Natsukawa is a project manager in the Software Innovation Center at Nippon Telegraph and Telephone Corporation (NTT), Tokyo, Japan. He received an M.E. from the Nara Institute of Science and Technology in 1996. He joined NTT in 1996. His current research interests include software engineering.



Hideya Iwasaki is a professor in the Graduate School of Informatics and Engineering at the University of Electro-communications. He has been a member of the Science Council of Japan since 2011. He received an M.E. degree in 1985 and a Dr. Eng. degree in 1988 from The University of Tokyo. His research interests include programming languages and systems, parallel processing, systems software, and constructive algorithmics. He is a member of the IPSJ and ACM.