UI X-Ray: Interactive Mobile UI Testing Based on Computer Vision

Chun-Fu (Richard) Chen
IBM T.J. Watson Research Center, USA
chenrich@us.ibm.com

Marco Pistoia
IBM T.J. Watson Research Center, USA
pistoia@us.ibm.com

Conglei Shi
IBM T.J. Watson Research Center, USA
shiconglei@us.ibm.com

Paolo Girolami*
IBM T.J. Watson Research Center, USA
University of Rome Tor Vergata, Italy
paolo.gyrolami@gmail.com

Joseph W. Ligman
IBM T.J. Watson Research Center, USA
jwligman@us.ibm.com

Yong Wang†
IBM T.J. Watson Research Center, USA
Hong Kong University of Science and Technology, Hong Kong
ywangct@cse.ust.hk

ABSTRACT

User Interface/eXperience (UI/UX) significantly affects the lifetime of any software program, particularly mobile apps. A bad UX can undermine the success of a mobile app even if that app enables sophisticated capabilities. A good UX, however, needs to be supported of a highly functional and user friendly UI design. In spite of the importance of building mobile apps based on solid UI designs, *UI discrepancies*—inconsistencies between UI design and implementation—are among the most numerous and expensive defects encountered during testing. This paper presents UI X-RAY, an interactive UI testing system that integrates computer-vision methods to facilitate the correction of UI discrepancies—such as inconsistent positions, sizes and colors of objects and fonts. Using UI X-RAY does not require any programming experience; therefore, UI X-RAY can be used even by non-programmers—particularly designers—which significantly reduces the overhead involved in writing tests. With the feature of interactive interface, UI testers can quickly generate defect reports and revision instructions—which would otherwise be done manually. We verified our UI X-RAY on 4 developed mobile apps of which the entire development history was saved. UI X-RAY achieved a 99.03% true-positive rate, which significantly surpassed the 20.92% true-positive rate obtained via manual analysis. Furthermore, evaluating the results of our automated analysis can be completed quickly (< 1 minute per view on average) compared to hours of manual work required by UI testers. On the other hand, UI X-RAY received the appreciations from skilled designers and UI X-RAY improves their current work flow to generate UI defect reports and revision instructions. The proposed system, UI X-RAY, presented in this paper has recently become part of a commercial product.

Author Keywords

User Interface Testing, Software Engineering, Interactive Interface

ACM Classification Keywords

D.2.5. Software Engineering: Testing and Debugging

INTRODUCTION

In the last few years, mobile computing has changed the way we interact with others and the world around us. This may be largely attributed to the application developer community and the creative way mobile applications help us achieve our daily tasks. As of June 2016, there are over 2 million Android and iOS apps available for download. With such a crowded marketplace, and an average usage length of around 1 minute, mobile applications will succeed only with a high-fidelity user interface and a strong design. Success typically requires a skilled design team that is differentiated from the engineering team developing the app. The engineering team must independently adhere to the UI design; design correctness is essential in mobile application development process.

A conventional UI testing flow in application development is illustrated in Figure 1, which is intensively manual. Developers start application development based on UI designs that designers created according to clients' requirements; at the same time, UI testers and designers define acceptance criteria for developers to test the functionality of applications and also describe what the UI implementations should be according to different interactions. It is easy for developers to assure the correctness of functionalities via unit tests, and there is always a numeric ground truth as a reference; however, UI verification is more subjective. Thus, it is usually done by skilled

*Paolo Girolami performed the work while as an intern at IBM Research.

†Yong Wang performed the work while as an intern at IBM Research.

IUI 2017, March 13–16, 2017, Limassol, Cyprus.

http://dx.doi.org/10.1145/3025171.3025190

UI testers and designers. As a result, developers might create correct functionality but inadvertently produce UI discrepancies. During testing, UI testers and designers prepare a UI defect report, which includes the locations of discrepancies along with revision instructions. This phase is usually extremely time consuming, since UI testers and designers need to manually label and accurately describe inconsistencies to communicate to developers where the defects are located and how to fix them. This process results in a communication latency between testers and developers, which significantly slows down application development. Furthermore, this process requires numerous iterations since UI testers and designers might not be able to find all the UI discrepancies at once by just eyeballing, and developers might even create more UI discrepancies during revision since not all defects are resolved for one view at the same time. UI discrepancies also take place when apps are upgraded, especially when the upgrade adds more functionality and new UI views. A robust regression UI testing is also time-consuming.

It should be noticed that developers have limited knowledge when it comes to fixing UI defects, and they might need to blindly try distinct strategies to fit the UI design requirements. In addition, there is no established toolkit like unit test to verify the UI implementations systematically. Thus, developers might accidentally commit new UI defects even while trying to fix existing ones. Also, UI testers and designers need to manually find UI discrepancies and generate UI defect reports for developers to guide them how to fix UI discrepancies. Both tasks are labor-intensive and time-consuming, without mentioning that the UI defect reports manually generated in the process above may be vague and not clearly guide developers. Finally, the smoothness of app development is also a concern, because UI testers might not be able to work on resolving UI discrepancies right after developers commit their codes. This causes developers to become idle between the time they commit their code and the time in which they receive revision instructions. The challenges we have just described become even more prominent when the app-development team is geographically distributed—a global trend in industry. In such cases, text descriptions are not suitable for resolving UI discrepancies.

In this paper, we propose UI X-RAY, an interactive UI testing system based on computer vision which unifies the UI testing environment for UI developers, testers and designers

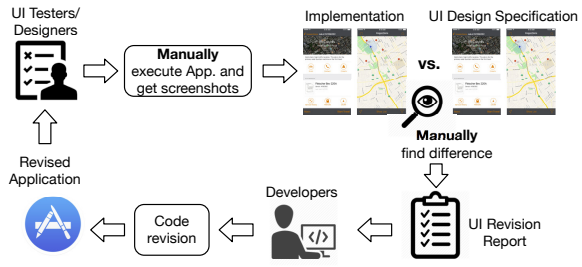

Figure 1. Conventional UI testing flow. Testers and designers need to manually interact with the app to find out UI discrepancies. Developers need to wait for the UI revision report before revising the app.

to align UI testing methods in finding and reporting UI discrepancies. A capable detection algorithm is not enough for resolving UI design discrepancies since many decisions are still made be experts for subjective evaluations; therefore, we integrated computer-vision algorithm with friendly interface design to achieve an intelligent and interactive UI testing system. UI X-RAY provides advantages from two aspects: (1) Developers have the capability to locally resolve as many UI discrepancies as possible thanks to the underlying computer-vision analysis, and they can also give feedback to UI testers and designers about the limitation of UI visualization in applications; and (2) UI testers and designers can quickly identify the remaining UI discrepancies based on the UI testing system and feedback from developers, and then provide revision instructions according to their expertise. Case 1 of Figure 2 illustrates the use case for developers. Developers can resolve the defects right away by utilizing the defect report produced by UI X-RAY. They can indeed fix as many discrepancies as possible before submitting their code. In parallel, UI testers and designers can enhance the report based on their expertise to guide developers to resolve those defects that might not be fixed by developers themselves, which is illustrated in case 2 of Figure 2.

This paper makes the following novel contributions:

1. We propose UI X-RAY, an interactive UI testing system that bridges the knowledge of UI testers, designers and developers, gives developers the power to find and fix discrepancies by themselves, and reduces the number of iterations necessary to fix discrepancies.
2. UI X-RAY is friendly to use for different types of users—UI testers, designers and designers.
3. UI X-RAY is based on a novel computer-vision algorithm to present UI discrepancies. UI discrepancies can be quickly detected and verified. In addition, UI X-RAY provides a friendly way for testers and designers to generate defect reports and revision instructions.
4. UI X-RAY's computer-vision analysis hierarchically reveals UI discrepancies, from a coarse-grained level to a fine-grained level. Our method does not just indicate the discrepancies but also suggests how to fix them.

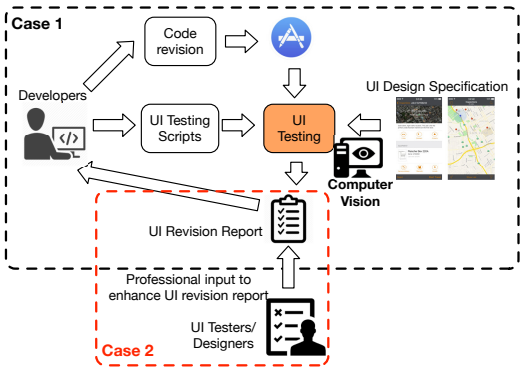

Figure 2. Improved UI testing flow based on UI X-RAY. Case 1: Developers are able to verify UI discrepancies by themselves, which shortens the time required to fix UI defects. Case 2: UI testers and designers can reuse the report created by developers to enhance UI revision reports.

5. UI X-RAY leverages the complexity of computer-vision analysis and the characteristics of UI design. Therefore, UI X-RAY can detect UI discrepancies efficiently and accurately, with a 99.03% true-positive rate that significantly surpassed the 20.92% true-positive rate obtained via manual analysis.
6. UI X-RAY has recently become part of a commercial product.

The remainder of this paper is organized as follows: we differentiate our novelty with related work. We then present the details of UI X-RAY along with its underlying computer-vision analysis and interactive interface design. We provide experimental results to justify the performance of UI X-RAY. Finally, we conclude the paper.

RELATED WORKS ON UI TESTING

This section describes related work in UI testing in comparison to UI X-RAY. In general, most of the research work in the field of UI testing is only about the detection algorithm and does not include a friendly interface to assist end users to easily interact with the results reported.

Fighting Layout Bugs [4] only evaluates the layout mismatch but does not identify different error types; on the contrary, UI X-RAY can categorize errors related to position, size, and color. Seleuium [12] is an automated testing tool for UI visualization of Web application. It requires a lot of user-input information to test the discrepancies. In contrast, UI X-RAY can automatically identify discrepancies based on computer-vision-based analysis.

Chang *et al.* [1] propose a Graphic User Interface (GUI) testing system with computer vision. They propose an element-level verification by examining the presence of graphic elements for corresponding behaviors. Nonetheless, their results can only check whether or not objects are present when the specific action is performed, and then raise a flag to indicate whether something is wrong. but fail to precisely identify where the problem is. Our work subsumes their capability since we check appearance of all elements, but in addition to their work, ours can locate UI discrepancies and provide revision insights.

Facebook [3, 11] has produced an open-source framework on element-level comparison to indicate UI difference/mismatch of elements from reference UI design, such as an icon, an image, and a button. Their work has been integrated in iOS and Android as a unit test system for development. Unfortunately, this tool requires deep knowledge of source code as well as programming experience, which decreases the usability by developers and UI testers. Furthermore, the Facebook work only compares single elements but ignores the relationship among elements, such as the relative positions of the various elements. Our work covers their capability and in addition suggests how to fix discrepancies in size, position, and color of elements.

Ligman *et al.* [6] propose a UI design-validation system based on application object-tree representation, screenshot, and object data. It uses view hierarchy and metadata of each view obtained by walking through source codes, and screenshots to validate the UI components against the design specification. Their results only check the correctness of individual UI elements, similar to the Facebook work discussed above; if a UI element appears in the wrong position, it might still pass the test as long as the UI element is correct. Furthermore, their work does not take into account background colors. Also, when an error occurs, there is no insight for developers to fix UI defects. Our method is a superset of their work because it compares not only UI elements, but also relationship between them. In addition, UI X-RAY also provides precise insights for resolving UI defects.

Mahajan *et al.* [8, 9] present a failure detection and localization of HTML errors, which utilizes HTML metadata to localize image differences and then report them to developers. Conversely, our approach detects, localizes, and measures defects only based on images. The presence of metadata would certainly be helpful, although not required. In addition, compared to UI X-RAY, which can measure the difference with pixel-level accuracy, they just produce coarse-grained results since they group defects into multiple clusters, which loses the capability for precise discrepancy measurement.

SYSTEM DESIGN

UI X-RAY is an interactive system that verifies UI implementations against the corresponding UI specifications by utilizing computer-vision algorithms. In this section, we first give an overview on how the system is designed, and then explain the algorithms in details.

Overview of UI X-RAY

Figure 3 shows UI X-RAY's workflow, which includes three modules: image retrieval, UI discrepancy identification, and an interactive report generation. For each UI view, the image-retrieval module is designed to automatically select the *specification* (the view drawn by UI designers) corresponding to the *implementation* (the view coded by the programmer). Since there are usually hundreds of design specifications per application, this module is essential because it automates an otherwise time-consuming activity. Having selected a specification/implementation pair, the UI discrepancy-identification module is used to indicate and measure any discrepancy through computer-vision-based analysis. The output from this module is used by the interactive report-generation module. This module provides a friendly interface, enabling end users to inspect, modify, and comment all the differences detected by the UI discrepancy-identification module. At this stage, a comprehensive and accurate UI defect report can be generated as a guidance to revise the implementation.

Image-retrieval Module

We designed and implemented a fast and efficient template-matching algorithm to find the most similar specification [5] for a given implementation. UI designs for mobile apps are usually more easily differentiable than natural images, which are composed of a variety of components and subject to lighting conditions, viewing angles, etc. Therefore, the intuition behind the design of this module is that a very complicated image retrieval algorithm is unnecessary. Traditionally, template matching is used to find the location of a subimage

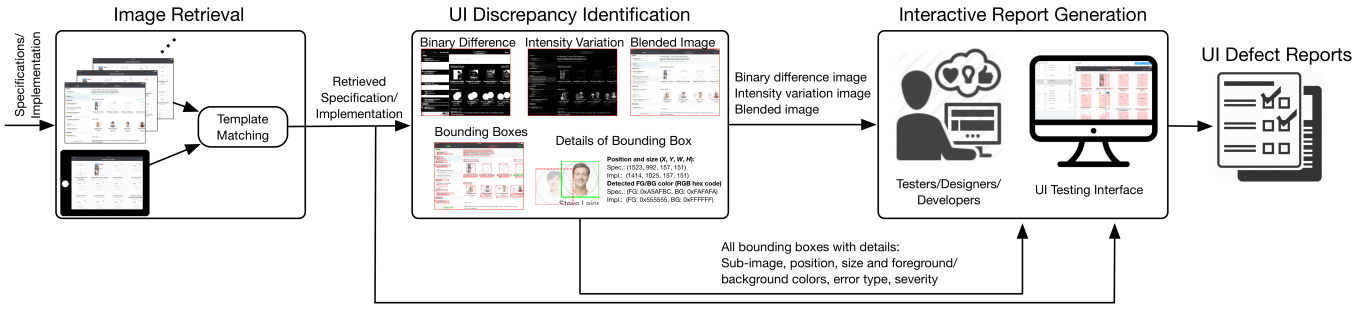

Figure 3. Workflow of proposed UI testing system, UI X-RAY

within an image, in which the detected subimage is the one most similar to the template, and the similarity is usually measured by the correlation. However, in our case, we consider the entire implementation as the template, and all specifications as a reference pool. In order to find the specification most similar to the implementation, we remodeled template matching by calculating the sum of absolute difference of the intensity value of every pixel in an image, and its expression is illustrated as Eq. 1:

$$Score(n) = \sum_{i,j} |Spec_I(n,i,j) - Impl_I(i,j)|, \quad \text{for } 0 \leq n < N \tag{1}$$

where $Score(n)$ is the matching score of the implementation to the n-th specification, $Spec_I(n,i,j)$ is the intensity I of pixel at position (i,j) in the n-th specification, $Impl_I(i,j)$ is the intensity I of pixel at position (i,j) in the implementation, and N is total number of specifications. Then, the specification with the smallest score will be retrieved. The next step is for the UI discrepancy-identification module to identify and measure the UI inconsistencies at the image and subimage levels.

UI Discrepancy-Identification Module

This section describes UI X-RAY's computer-vision-based analysis, while Figure 4 displays an example. There are two stages to identify and measure UI discrepancies. First, we identify them at the image level to form a big picture and investigate suspicious regions; then, we decompose an image into subimages by detecting corresponding bounding boxes to find details and provide revision insights to fix defects, such as size, position and color. Thus, users can hierarchically identify issues from a coarse-grained view to a fine-grained view instead of being overwhelmed by plentiful information at the same time. The following two subsections describe the image-level and subimage-level analyses respectively.

Image-Level Analysis

Through image-level analysis, we identify UI discrepancies in three different formats; each one reports discrepancies from different perspectives: (I) binary difference, (II) intensity variation, and (III) blended image—as shown on the top of Figure 4.

1. **Binary difference**: This image reports whether or not the pixels in colocated positions in the implementation and the corresponding n-th specification previously retrieved are identical. A white pixel denotes a mismatch, a black pixel a match. Equation 2 illustrates how the binary difference image is derived.

$$B(n,i,j) = \begin{cases} 255, & Spec_I(n,i,j) \neq Impl_I(i,j) \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where $B(n,i,j)$ is the pixel value of binary difference image at position (i,j). Thus, users can easily delineate the suspicious regions and remove false-alarm regions. For example, in Figure 4, the region in the middle is a suspicious area (circled by a red-dashed box), which indicates a mismatch in both text and background.

2. **Intensity variation**: This image indicates differences between the implementation and the n-th retrieved specification, where each difference is displayed proportionally to the difference of pixel intensity. To derive the intensity

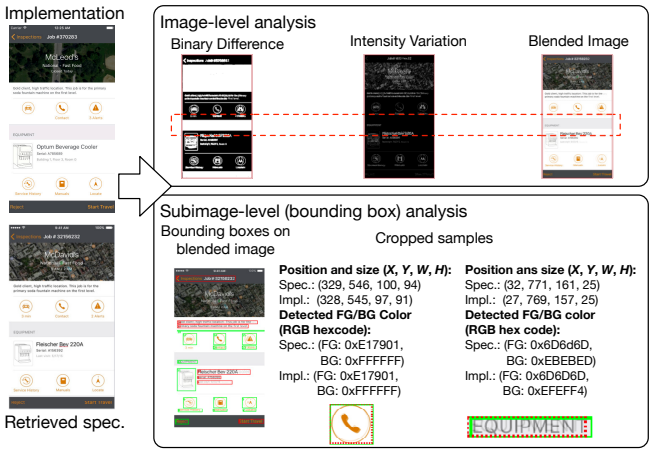

Figure 4. Details of UI X-RAY's UI discrepancy-identification module, including both image-level and subimage-level (bounding-box) analysis. At the image level, three different visual representations are shown to identify locations of discrepancies. At the subimage level, all UI elements are detected and individual tests on each box are performed to examine discrepancies at a fine-grained level. Green boxes denote passed tests and red boxes failed tests. Each bounding box is a cropped area of the blended image in which the green box is the specification and the red box with dashed line is the implementation. Furthermore, we measure bounding box information of each cropped area as metadata, including the X and Y coordinates of the top-left corner of the bounding box, width W, height H, and the associated foreground (FG) and background (BG) colors.

variation map, we apply Eq. 3 on every pixel pair:

$$I(n,i,j) = |Spec_I(i,j) - Impl_I(i,j)| \quad (3)$$

where $I(n,i,j)$ is the value of intensity variation for pixel at position (i,j) when comparing the implementation to the n-th retrieved specification. The intensity-difference image is likely to reveal more details compared to the binary-difference one. Even though users can quickly spot differences using the binary approach, the binary-difference image may hide important information. Furthermore, the intensity-difference image may help us distinguish defect types. For example, for the same white region in the binary-difference image of Figure 5, it is only by examining the intensity-variation image that we learn that the difference comes not only from the background color but also from the text misalignment.

3. **Blended image**: This image blends the specification and the implementation into one single image through alpha blending. The specification weighs more than the implementation in order to guide the user to distinguish the specification from the implementation. For example, the word "EQUIPMENT" in the implementation is shifted to the left compared to the specification. We can fix this by moving the text box to right by the correct number of pixels. The blended image can be derived by Eq. 4:

$$Blend(n,i,j) = \alpha Spec(n,i,j) + (1-\alpha)Impl(i,j) \quad (4)$$

where $Blend(n,i,j)$ is the value of the pixel at position (i,j) in the image that blends the implementation with the n-th specification, $Spec(n,i,j)$ is the n-th retrieved specification with RGB components and $Impl(i,j)$ is the implementation with RGB components. We set α to 0.7 in our experiments.

Subimage-Level (Bounding Box) Analysis

The image-level analysis *identifies* areas of discrepancy. The subimage level measures those discrepancies and gives users insights on *how to fix* them.

First, UI X-RAY decomposes the whole image into multiple subimages by detecting salient regions. Each region is surrounded by a rectangular bounding box; in mobile applications, salient regions typically are text fields, objects, tables, and buttons. Each bounding box is associated with the X and Y coordinates of the top-left corner, as well as width W and height H. Due to natural UI design in mobile applications, a bounding box, which surrounds a fine-grained UI element, usually contains only two major colors, foreground and background. UI X-RAY can identify the foreground and background colors of a bounding box. Foreground colors have different meanings depending on the bounding box type. For example, in a text bounding box the foreground color is the font color, while in an object bounding box the foreground color is the color of the main object. Furthermore, the height of a text bounding box can be also used to infer the font size. After having independently identified bounding boxes in the specification and the implementation, we pair them based on their positions; therefore, we measure their dissimilarity with pixel-level accuracy.

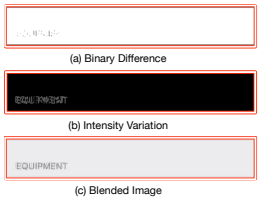

Figure 5. Cropped patches of red rectangular in Figure 4

A fundamental characteristic of mobile UIs is the fact that in mobile applications, the regions of interest are characterized by high contrasts. This allows UI X-RAY to apply edge detection to accurately identify salient points, such as edges of texts and icons. Afterwards, morphological methods, dilation and erosion, are used to connect sparse salient points in forming connected components. Thus, we can delineate those connected components by bounding boxes to obtain subimages [10]. Each detected bounding box usually encloses content of a single type, such as a text field or an icon, as shown in the green and red boxes at the bottom of Figure 4.

We can also detect foreground and background colors of each bounding box based on k-means clustering [7] since each bounding box, as we just mentioned, encloses contents of a single type, which results in the color contrast to be clear. For example, contrast of text and background is high in order to make users read text more easily. This allows us to choose the two most frequent color components in each cluster to be foreground and background colors, respectively. Therefore, we are able to report UI discrepancies on colors of font, icons, and backgrounds.

To correlate bounding boxes in both images, we deploy the Non-Maximum Suppression (NMS) method to retrieve the pairs of the most correlated subimages in two different images. NMS measures the ratio of intersected area over union area of two bounding boxes as defined in Eq. 5. A higher NMS value denotes higher bounding box similarity.

$$NMS_i = \underset{j}{\text{argmax}} \frac{Area_{bbox_i \cap bbox_j}}{Area_{bbox_i \cup bbox_j}} \quad (5)$$

where $bbox_i$ is the i-th bounding box in the specification and $bbox_j$ is the j-th bounding box in the implementation.

We present the above results by superimposing bounding boxes on the blended image; hence, we are able to find the discrepancies at both the image and subimage levels concurrently. Next, we crop those bounding boxes into individual subimages, each with its bounding box information, including the bounding box's top-left corner's X and Y geometrical coordinates, width W, height H, and foreground and background colors, for both the specification and the implementation.

Based on this precise discrepancy measurement, developers are now guided to resolve those defects by comparing the difference of the information of bounding boxes. For example,

in Figure 4, the exact difference of the phone icon is difficult to identify by plain eyeballing, but with the aid of the surrounding bounding box, UI testers and designers can learn that there is a one-pixel mismatch between the implementation and the specification in both the X and Y coordinates, while the implementation's size is 3 pixels smaller than the specification in both width W and height H. Furthermore, in Figure 6, our method also points out the discrepancies in font and background colors, which are difficult to find manually. Foreground color discrepancy is used to fix font or icon color, whereas background color inconsistency indicates that the color of the entire background was not implemented according to the specification. In Figure 6 (a), the font color discrepancy is easier to find when we enlarge and focus on the subimage. However, when the inconsistency involves a piece of whole image or its background, automatically detection and reporting discrepancies is more difficult. In subfigure (b) of Figure 6, the cyan color of text "View All" looks very similar but their hex color codes are different. Figure 6 (c) indicates that the background color is wrong, which is also related to the discrepancy we found in Figure 5, with the difference that we have now obtained more insights to fix the color issue. Font size can be inferred from the height of the bounding box. For Figure 6, since those applications are rendered at $2\times$ scale, we can infer the font size by dividing the height by 2. Thus, font sizes of (a), (b), and (c) are 17, 15 and 17, respectively (Some letters exceed the baseline, so we round down to the integer.).

It may not be realistic to always demand the implementation to be perfectly identical to the specification. For example, designers might not know particular programming limitations that cause the specification to become unimplementable. Another situation that calls for accepting the implementation over the specification takes place when the bandwidth of developers is so tight that slight mismatches must be deemed to be acceptable. Having attained precise discrepancy measurement, we can set up a *tolerance coefficient* to report only severe discrepancies that fall beyond that coefficient. Testers and designers can tell developers how to adjust the acceptance criteria by setting the tolerance coefficient accordingly.

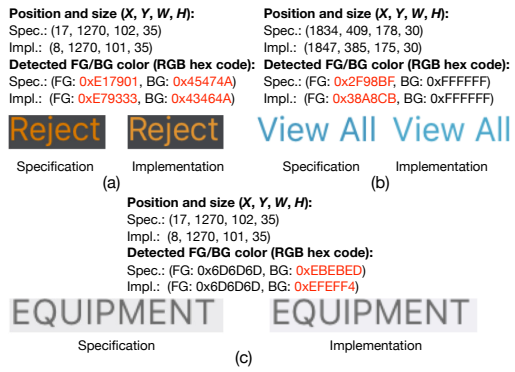

Figure 6. Three examples of font foreground (FG) and background (BG) color detection

We provide flexible tolerance setting of bounding boxes in position, size and color. Two parameter sets are included in tolerance setting: position tolerance (ΔX, ΔY, ΔW, ΔH), and color tolerance (ΔR, ΔG, ΔB). Figure 7 shows three different position tolerance settings on (ΔX, ΔY, ΔW, ΔH) and the corresponding details of each bounding box. No tolerance setting is equivalent to the strictest setting, which disallows any mismatch, while a tolerance coefficient of -1 indicates infinite tolerance. Tolerance setting A relaxes the tolerance on the Y coordinate and sets it to infinity, while the other parameters have a coefficient tolerance of 4. Tolerance setting B completely relaxes both the X and Y coordinates and just verifies the size of the bounding boxes by imposing that any mismatch in width and height be smaller than 4 pixels. As a result, Figure 7 (a) corresponds to three test failures because there are three mismatches and no tolerance for any discrepancy. Figure 7 (b) gets one pass and two failures since the bounding box in the center fits the tolerance criteria but two boxes at the sides do not. Finally, Figure 7 (c) passes all tests because the size difference is 3 pixels but tolerance is 4 pixels. Similar reasoning can be applied to color tolerance. For example, in Figure 6 (a), the background color is close but slightly different, so we can set color tolerance (ΔR, ΔG, ΔB) to (2, 1, 0) if the background color of the implementation is acceptable.

This tolerance setting also resolves issues about dynamic data, since sometimes the examples used in specifications will be different from real data used in implementations, e.g. The equipment names of specification and implementation in Figure 4 are difference; then, the width of detected bounding box would be different. Thus, by setting ΔW to -1, we can still validate the height, as well as the X and Y coordinates to assure that the implementation conforms the specification.

The above verification logic provides the insights to fix discrepancies. We can also extend UI X-RAY to validate the implementations in regression testing, and test scripts written by developers can be reused to assure that existing UIs do not violate the previous tolerance when adding new UIs or functionalities.

Interactive Report Generation Module

The detected differences from the aforementioned modules are not always accurate. For example, by comparing the map at the top of the specification and implementation images

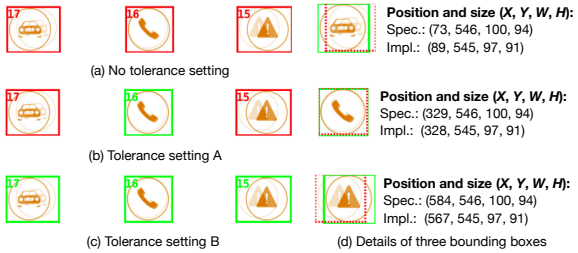

Figure 7. Flexible tolerance (ΔX, ΔY, ΔW, ΔH) for examining bounding boxes. (a) No tolerance setting. (b) Tolerance setting A = (4, -1, 4, 4). (c) Tolerance setting b = (-1, -1, 4, 4). (d) Details of the bounding boxes.

in Figure 4, we see a mismatch, which is also reported by the binary, intensity and blended images. This mismatch is, however, a false positive. The designer used a map centered around McDavid's National Fast Food, while the data dynamically retrieved during testing brought up an implementation with a map centered around McLeod's National Fast Food. This mismatch can safely be ignored because it is normal for the app to work on dynamically generated data, which may not correspond to the sample data used during design. This shows that, in some cases, additional comments may be needed for developers to better understand how to revise an implementation correctly. The interactive report-generation module fills the gap between the result generated by the UI discrepancy-identification module and the final UI defect report.

Designing this module was an iterative process. We collaborated with several designers and had multiple rounds of meetings with them to better understand their needs for such tool to be useful. Below is the summary of the requirements we gathered:

- **Order matters**. Since a considerable number of differences will be detected, it is necessary to present all of them in an organized manner.
- **Better comparison is needed**. End users need to verify all the detected differences between an implementation and the corresponding specification. In order to speed up this process, a better comparison approach is needed for them to quickly and accurately conducting comparisons between two images.
- **Modification and commenting is a must**. As mentioned before, it is not guaranteed that the differences detected automatically are always correct—false positives are possible. Therefore, manual modification of the differences is necessary. Furthermore, in order to give detailed and precise instructions on how to revise the implementation, commenting on any detected difference is also a necessary feature.

Guided by these requirements, we developed a Web application, shown in Figure 8. The application consists of two views: a table view listing all the differences, and a sliding view providing a straightforward in-situ comparison of the detected differences. For each bounding box, we compute four type of discrepancies: position (X, Y), size (W, H), foreground color $FG(R, G, B)$ and background color $BG(R, G, B)$. A bounding box's *severity* is computed as the sum of problems associated with it. UI X-RAY lists the discrepancies in the table view ordered by severity. Furthermore, for each discrepancy, the error types and the corresponding thumbnail are provided in the table for users to quickly identify the position of that discrepancy and its severity. The bounding boxes—the one detected in the specification, the corresponding one detected in the implementation, and the union of the two—are rendered in the sliding view together with the images of original implementation and specification, as shown in Figure 9. The two images will be overlaid and the divider (in blue color) is used to distinguish the implementation with specification. For example in Figure 9 (b), the divider is dragged in the middle, where the left-hand side shows the implementation and the right-hand side the specification. In this way, users can easily compare the two corresponding regions. The union bounding box will be rendered as a rectangle with no fill color but with a gray border line. For the other two bounding boxes, they will be only rendered as a pink rectangle on the opposite images (i.e., the bounding box detected from the image of the implementation will be rendered on the image of the specification, and vice versa). This rendering strategy enables users to quickly identify the difference between the implementation and specification without directly comparing both images. Modification and commenting are supported by selecting a specific difference either through the table view or the sliding view.

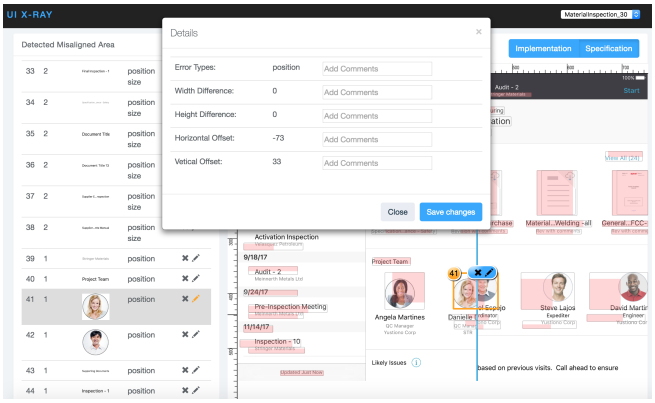

Figure 8. Screen shot of the interactive user interface. The Web application contains a table view and a sliding view. For each difference, users can either remove it or add more comments through a pop-up window.

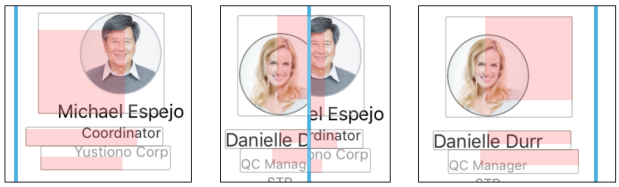

Figure 9. The illustration of sliding interaction. (a) the image from the specification; (b) the mixture of images from the implementation and specification; and (c) the image from the implementation.

Usage Scenario

To help better understand the interactive report generation module, we illustrate it using a scenario with a designer, Amy, trying to use UI X-RAY to generate an UI defect report.

After developers finished implementing the UI she designed, Amy needs to inspect the implementation and give feedback to the developers on how to revise the implementation. Having automatically gotten the screen shot of the implementation, the image retrieval module selects the corresponding specification, and these two are used as the input to the UI discrepancy-identification module. After all of the discrepancies, along with their bounding boxes, are detected and rendered by the interactive report-generation module, Amy can use this module to inspect all the differences.

Starting from the table view, Amy is able to have a brief look of the differences according to their severity. She can mouse over each row in the table to see the corresponding highlighted area in the sliding view. By inspecting the the pink area, Amy can have an initial idea about the reported discrepancy. If it is a false alarm, she can just delete the difference by clicking the **delete** icon. In order to further compare the implementation and the specification, Amy can drag the divider to the highlighted region and compare the corresponding images. If this difference is confirmed, Amy can click the **edit** icon to review the quantitative value of the difference. If needed, she can add more comments on each error type and save the changes. After iterating the process for all the discrepancies, the result will be saved and can be used as the UI defect report for developers to revise the implementation accordingly.

SYSTEM EVALUATION

To evaluate UI X-RAY, we performed a quantitative analysis. We also collected reviews by experts to verify the effectiveness and usefulness of UI X-RAY.

Quantitative Evaluation of UI X-RAY

Even though the method behind UI X-RAY is general and can be applied to different programming environments, we integrated UI X-RAY's computer-vision-based analysis into the iOS application development ecosystem in the form of unit tests, and evaluated UI X-RAY on 4 fully developed iOS mobile applications of which the entire development history was saved.

The performance metric is measured by comparing the correctness of the UI discrepancies detected by UI X-RAY vs. those manually reported by UI testers and designers. Another dimension of comparison involves the time necessary to detect and fix the UI discrepancies detected. Since we had access to the full history of the developed apps, we were able to reproduce all the intermediate versions of each app, examine the UI defects that were manually detected by the development team, compare them with those detected by UI X-RAY, and measure the time necessary to fix those discrepancies using manual analysis vs. UI X-RAY. The apps we had access to were developed in Agile mode.

Our evaluation consisted of the following steps:

1. Locate in the development history the precise point in which developers were notified of the UI defects and asked to fix them—this happened at the end of each sprint
2. Go back to the intermediate version of the app right before the fixes were applied
3. Execute UI X-RAY on that version to compute all the UI defects on that version of the app
4. Numerically compare the results obtained via manual methodology vs. UI X-RAY
5. Compare the time involved in detecting and fixing all the UI discrepancies using the two methodologies

Thanks to record-and-run technologies in modern mobile-app development for UI testing—such as UI recording in Xcode [13] and Espresso Test Recorder in Android Development Tools [2]—the test scripts can be automatically generated by recording the app execution. Record-and-run provides a fascinating feature to automatically generate the UI testing scripts based on the user's interaction with the app. It is then sufficient to insert a call to the UI X-RAY API in order to invoke the UI discrepancy detection routine every time the underlying operating system generates a new view for the app. The overall detection results are summarized in Table 1.

Table 1. UI discrepancies detection results of manual analysis and UI X-RAY.

App	True Positives		False Negatives	
	Manual Analysis	UI X-RAY	Manual Analysis	UI X-RAY
App 1	26	135	109	0
App 2	10	61	51	0
App 3	93	699	606	0
App 4	131	336	217	12
Total	260	1,231	983	12

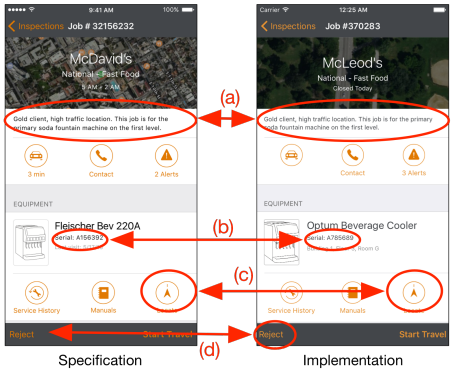

Figure 10. Experimental results of App 1

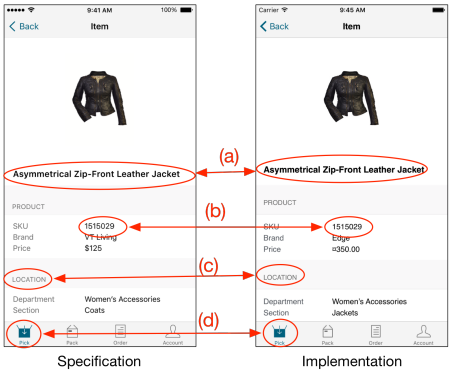

Figure 11. Experimental results of App 2

The true-positive rate achieved by UI X-RAY is 99.03% vs. 20.92% obtained via manual analysis. In total, on the four apps we examined, UI X-RAY detected 983 real issues that were not discovered by manual analysis in spite of the fact that manual analysis was significantly more expensive in terms of people involved and time spent.

Table 2. Comparison details of Figure 10, 11, 12 and 13

		Position and Size (X, Y, W, H) Spec.	Position and Size (X, Y, W, H) Impl.	FG/BG Colors (RGB hexcode) Spec.	FG/BG Colors (RGB hexcode) Impl.
Figure 10	(a)	(31, 438, 612, 26)	(26, 439, 666, 26)	FG: 0x000000 BG: 0xFFFFFF	FG: 0x43464A BG: 0xFFFFFF
	(b)	(193, 902, 196, 22)	(194, 903, 189, 22)	FG: 0x191919 BG: 0xFFFFFF	FG: 0x45474A BG: 0xFFFFFF
	(c)	(586, 1082, 100, 94)	(567, 1083, 97, 91)	FG: 0xE17901 BG: 0xFFFFFF	FG: 0xE17901 BG: 0xFFFFFF
	(d)	(17, 1276, 102, 35)	(8, 1276, 101, 35)	FG: 0xE17901 BG: 0x45474A	FG: 0xE79333 BG: 0x43464A
Figure 11	(a)	(33, 646, 660, 36)	(28, 630, 636, 36)	FG: 0x000000 BG: 0xFFFFFF	FG: 0x000000 BG: 0xFFFFFF
	(b)	(313, 852, 124, 26)	(298, 855, 122, 25)	FG: 0x000000 BG: 0xFFFFFF	FG: 0x000000 BG: 0xFFFFFF
	(c)	(31, 1062, 137, 23)	(28, 1053, 137, 24)	FG: 0x6D6D6D BG: 0xF9F8F9	FG: 0x6D6D6D BG: 0xF9F8F9
	(d)	(46, 1249, 71, 53)	(61, 1251, 67, 50)	FG: 0x00617E BG: 0xF4F4F4	FG: 0x00617E BG: 0xF4F4F4
Figure 12	(a)	(214, 70, 213, 35)	(214, 70, 214, 36)	FG: 0xFFFFFF BG: 0x374B58	FG: 0xFFFFFF BG: 0x223842
	(b)	(22, 192, 235, 26)	(30, 197, 236, 26)	FG: 0xC2CCD4 BG: 0x3E5564	FG: 0xC2C2C2 BG: 0x283740
	(c)	(31, 645, 142, 29)	(31, 646, 136, 29)	FG: 0xFFFFFF BG: 0x688699	FG: 0xFFFFFF BG: 0x39424A
	(d)	(22, 1036, 116, 26)	(30, 1037, 116, 26)	FG: 0xC2CCD4 BG: 0x496576	FG: 0xC2C2C2 BG: 0x233039
Figure 13	(a)	(917, 178, 217, 36)	(917, 178, 219, 36)	FG: 0xFFFFFF BG: 0x31302F	FG: 0xFFFFFF BG: 0x545456
	(b)	(798, 686, 248, 84)	(850, 682, 248, 84)	FG: 0xFFFFFF BG: 0xFC7409	FG: 0xFFFFFF BG: 0xFD7409
	(c)	(890, 715, 63, 28)	(947, 712, 55, 26)	FG: 0xFFFFFF BG: 0xFC7409	FG: 0xFFFFFF BG: 0xFD7409
	(d)	(521, 887, 196, 23)	(512, 884, 191, 23)	FG: 0x6D6D6D BG: 0xF5F4F0	FG: 0x6D6D72 BG: 0xEFEFF4

The number of reported UI defects of each app vary based on the length of the development lifecycle, distinct UI design characteristics, and different UI testers. Therefore, the number of additional true positives detected by UI X-RAY may differ significantly app by app. For example, in apps 1 and 2, the UI designs are icon- and item-driven and contain much fewer text fields. This implies that most of defects are the misalignment of objects or inconsistent sizes. As a result, the number of additional true positives are close to each other for these two apps, and less than what reported for apps 3 and 4. On the other hand, apps 3 and 4 contain more tables and forms, which usually cause UI defects on misalignment of table header and cells, inconsistent background colors and font size, etc. Therefore, UI X-RAY detects more true positives in apps 3 and 4 than it does in apps 1 and 2.

In terms of *time*, our analysis of the Agile development data—diligently stored by the development team using IBM Rational Collaborative Lifecycle Management (CLM)—revealed the following important points:

1. On average, UI defects constitute 60% of the total number of defects of a mobile app.
2. At the end of each sprint:
   - The development team lead spends approximately 1.5 days to manually detect and report the UI inconsistencies on the intermediate version of the app
   - *Each* developer in the team spends approximately 2 days to interpret the development team lead's report, fix the UI, test it, and communicate back and forth with the development team lead until the UI bugs are deemed fixed

On average, each of the app we examined was developed by a team of 6 developers, not including the development team lead, and the entire development lifecycle consisted of 5 sprint. Overall, fixing the UI bugs during the development of each app required $1.5 \times 5 + 2 \times 5 \times 6 = 67.5$ man days. Conversely, UI X-RAY allows for integrating the UI detection and repairing system directly into the app lifecycle, thereby making it possible for developers to immediately detect any UI discrepancy and fix it on the fly with virtually no impact on the overall development time. Specifically, UI X-RAY does not only provide accurate detection of UI discrepancies, but it can also be executed quickly: less than 1 minute per view. In traditional manual testing, UI defects are reported and fixed one by one, and this type of procedure is quite inefficient. Instead, UI X-RAY locates multiple bugs in a single test, reports them straightforwardly on the provided comparison images, and simultaneously records the differences on each subimage. Hence, developers can resolve all related defects on the same image at once, and save time.

Another important point to notice is the contribution that UI X-RAY provides in terms of *precision*. The description of the UI inconsistencies provided by the development test lead is reported in CLM in the form of a textual message, which is often vague and ambiguous. For example: "*Fonts are too large*"—which fonts in the view is the development team lead referring to, and how larger are those fonts?—"*Color is darker than it should be*"—which color is the development team lead referring too, and how darker is it? UI X-RAY completely resolves this ambiguity by bypassing the need for textual messages. UI X-RAY uses image recognition to automatically detect the UI design in the design document—which often contains hundreds of images—thereby saving both the development team lead and the programmers from the burden of having to locate the design specification corresponding to each view. Furthermore, the reports generated by UI X-RAY visually highlight and numerically quantify the UI inconsistencies between the UI specification and the corresponding implementation. Thanks to this feature, the need for manually describing UI defects and communicating them is completely bypassed.

We display subjective results of each application from Figure 10 to 13. We just show 4 representative defects in each example. All the details on the corresponding discrepancies are illustrated in Table 2.

Figure 10 (a) and (b) indicate the difference on the font color. For example, the specification requires black but the implementation uses gray instead; Figure 10 (c) points out the discrepancies of position and size, and it also implies the left-margin to the center icon is wrong. The location of that icon should be moved right by 9 pixels and the icon size should be reduced by 3 pixels. Figure 10 (d) detects defects in both font and background colors.

Figure 11 shows multiple mismatched left margins in text field (a), table content (b), table section header (c), and buttons (d). UI X-RAY also points out a discrepancy in Figure 11 (b): the size of the table cells is not assigned well, which results in the misalignment. A mismatch at Figure 11 (d) im-

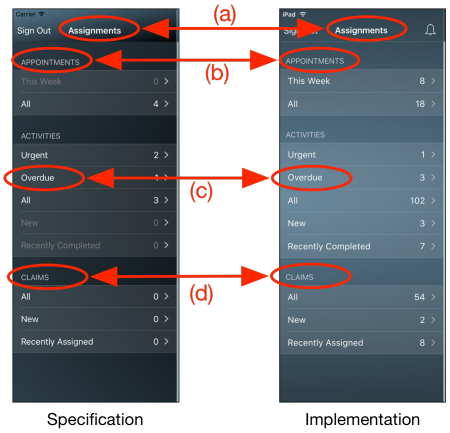

Figure 12. Experimental results of App 3

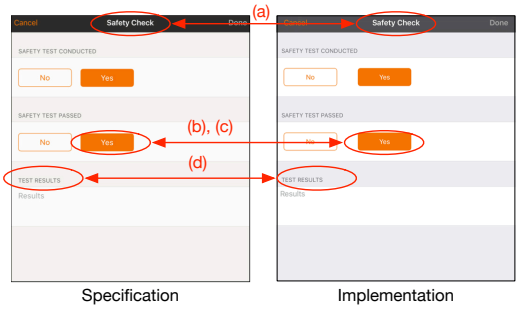

Figure 13. Experimental results of App 4

plies that the distance between buttons must be investigated in order to derive correct position.

Figure 12 shows the discrepancies in a table view. Four defects all indicate that the background colors are wrong since the specification use darker colors. For the table section headers at (b) and (d), there is an 8-pixel displacement and an inconsistency in the font color. As for the table content, there is no position issue at (c).

Figure 13 (a) indicates the defect in the background color of the table header; Figure 13 (b) points out that the position of the **Yes** button is too far from the one of the **No** button; developers would convert the distance to the margin between the two buttons. Figure 13 (c) reports that the height of the bounding boxes surrounding the **Yes** text are different, which implies that the font sizes are different, too: 14 font size should be used; last, Figure 13 (d) marks that the left-margin of text region is too small and the font and background color in the implementation are wrong.

Expert Interviews

We conducted in-depth interviews with three UI/UX designers of a company to validate the usability of UI X-RAY. One designer (D1) was primarily working on Web application design but also had some mobile app design experience. The other two designers (D2, D3) are iOS app design experts. All the interviews started with several questions to identify their background and how they did UI testing before. After that, we gave them a tutorial showing how UI X-RAY worked, letting them use their own computers to try UI X-RAY. In the end, we asked them post-study questions to gather their feedback and suggestions. Each interview lasted between 30 and 60 minutes.

Overall System Usability and Interaction Design

All the designers spoke highly UI X-RAY. Since there was no tool before to help them compare an implementation with the corresponding specification, they had all relied on "eyeballing" the two images and use some graphic design tools such as Sketch or Adobe Illustrator to make the comparison. All of them stated that UI X-RAY dramatically speeds up the comparison job and significantly adds precision to the testing process. D3 commented: "*I think this is really great. I do everything manually now; I type the notes, record everything and identify all the little spots, and that takes a lot of time.*" The sliding and comparing functionality received the most appreciation: "*The swiping interaction is really cool*" (D3), "*I like the sliding feature a lot since it allows me to directly compare two regions with little occlusion problems*" (D1), and "*The sliding interaction can greatly help me to quickly compare my design with the implemented version*" (D1). Moreover, they all agreed that the modification and commenting features were very important for them to finalize the revision plan, and D3 further commented: "*This tool would be very helpful if a team is not geographically located at the same site.*".

Suggestions

During the 3 interviews, we also collected several suggestions on how to improve UI X-RAY. So far, for each inconsistency, only the differences in color, position and size are reported. However, sometimes, developers and designers may have different opinions on what should be reported. For instance, in a grid component, the margin between elements is the most useful value. More specifically, D2 commented the following: "*Implementation-wise, different visual parameters, such as paddings and margins, will be used to decide the position of an element. Therefore, it would be nice if the system could provide the differences of these parameters according to the nature of an element.*"

Another suggestion common to all of the designers is that adding the comparison of interaction behaviors will be very helpful to see, for example, whether a button clicking behavior is as expected. The designers also had some other suggestions. Specifically, D3 pointed out that it would be useful to check the static content for typos, while D1, who works on Web application design as well, suggested that responsive designs (i.e., the layout can be adjusted according to the resolution of the window size) should also be considered when conducting comparisons.

CONCLUSION AND FUTURE WORK

In this paper, we presented UI X-RAY, an interactive UI testing system to resolve labor-intensive work in finding and fixing UI discrepancies. UI X-RAY features a user-friendly interface to inspect, annotate, and comment the UI discrepancies found by underlying computer-vision based analysis. UI X-RAY achieves good outcomes in both quantitative and

qualitative evaluations; in quantitative evaluation, UI X-RAY exhibits a 99.03% true-positive rate, which significantly surpasses the 20.92% true-positive rate obtained by manual analysis. Furthermore, every evaluation and correction when using UI X-RAY is done within 1 minute on average vs. hours required when using manual analysis. In qualitative evaluation, UI X-RAY received extremely positive responses from professional designers, who reported how UI X-RAY's interactive interface can help them find and comment on discrepancies quickly. Furthermore, UI X-RAY assists programmers to fix the majority of the UI discrepancies as soon as they arise during development, which significantly reduces the workload on UI testers and designers. Specifically, UI X-RAY reduces the number of iterations necessary to fix UI discrepancies, and virtually eliminates the need for communication between developers and UI testers and designers. UI X-RAY has recently became part of a commercial product.

UI X-RAY includes powerful capabilities to verify UI representation and report detailed instructions on how to fix inconsistent positions, sizes and colors of objects and fonts. As part of future work, we are planning to enhance UI X-RAY with the ability to report inconsistent font face, perform static content verification (for example, to detect typos in static text), and execute behavior-based verification.

REFERENCES

1. Chang, T.-H., Yeh, T., and Miller, R. C. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (New York, NY, USA, 2010), 1535–1544.
2. Espresso Test Recorder. `http://android-developers.blogspot.nl/2016/05/android-studio-22-preview-new-ui.html`, 2016.
3. FBSnapshotTestCase. `https://github.com/facebook/ios-snapshot-test-case`, 2016.
4. Fighting Layout Bugs. `https://code.google.com/archive/p/fighting-layout-bugs/`.
5. Lewis, J. P. Fast template matching. In *Vision interface*, vol. 95 (1995), 15–19.
6. Ligman, J., Pistoia, M., Tripp, O., and Thomas, G. Improving Design Validation of Mobile Application User Interface Implementation. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ACM (New York, NY, USA, 2016), 277–278.
7. Lloyd, S. Least squares quantization in pcm. *IEEE Trans. Inf. Theor. 28*, 2 (Sept. 2006), 129–137.
8. Mahajan, S., and Halfond, W. G. J. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE (Apr. 2015), 1–10.
9. Mahajan, S., Li, B., Behnamghader, P., and Halfond, W. G. J. Using Visual Symptoms for Debugging Presentation Failures in Web Applications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE (Apr. 2016), 191–201.
10. Open Source Computer Vision Library. `http://opencv.org`, 2016.
11. Screenshot Tests for Android. `https://github.com/facebook/screenshot-tests-for-android`, 2016.
12. Selenium. `http://docs.seleniumhq.org/`.
13. User Interface Testing. `https://developer.apple.com/videos/play/wwdc2015/406/`, 2015.