Automated Visual Testing for Mobile Apps in an Industrial Setting

Dezhi Ran
School of Computer Science, Peking University, China
dezhiran@pku.edu.cn

Zongyang Li
School of Software and Microelectronics, Peking University China
lizongyang@stu.pku.edu.cn

Chenxu Liu
School of Computer Science, Peking University, China
lcx@stu.pku.edu.cn

Wenyu Wang
University of Illinois Urbana-Champaign USA
wenyu2@illinois.edu

Weizhi Meng
Alibaba Group, China
weizhi.mwz@alibaba-inc.com

Xionglin Wu
Alibaba Group, China
xionglin.wxl@alibaba-inc.com

Hui Jin
Alibaba Group, China
dingyuan.jh@alibaba-inc.com

Jing Cui
Alibaba Group, China
qingling@taobao.com

Xing Tang
Alibaba Group, China
pingchou.pc@alibaba-inc.com

Tao Xie*
School of Computer Science, Peking University, China
taoxie@pku.edu.cn

ABSTRACT

User Interface (UI) testing has become a common practice for quality assurance of industrial mobile applications (in short as apps). While many automated tools have been developed, they often do not satisfy two major industrial requirements that make a tool desirable in industrial settings: high applicability across platforms (e.g., Android, iOS, AliOS, and Harmony OS) and high capability to handle apps with non-standard UI elements (whose internal structures cannot be acquired using platform APIs). Toward addressing these industrial requirements, automated visual testing emerges to take only device screenshots as input in order to support automated test generation. In this paper, we report our experiences of developing and deploying VTest, our industrial visual testing framework to assure high quality of Taobao, a highly popular industrial app with about *one billion* monthly active users. VTest includes carefully designed techniques and infrastructure support, outperforming Monkey (which has been popularly deployed in industry and shown to perform superiorly or similarly compared to state-of-the-art tools) with 87.6% more activity coverage. VTest has been deployed both internally in Alibaba and externally in the Software Green Alliance to provide testing services for top smartphone vendors and app vendors in China. We summarize five major lessons learned from developing and deploying VTest.

CCS CONCEPTS

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

*Tao Xie is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China, and is the corresponding author.

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513027

KEYWORDS

UI testing, visual testing, mobile testing, robotic testing

ACM Reference Format:
Dezhi Ran, Zongyang Li, Chenxu Liu, Wenyu Wang, Weizhi Meng, Xionglin Wu, Hui Jin, Jing Cui, Xing Tang, and Tao Xie. 2022. Automated Visual Testing for Mobile Apps in an Industrial Setting. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513027

1 INTRODUCTION

With the flourish of mobile devices, people have been increasingly dependent on these devices [40] in their daily work and life. Ensuring good user experiences of mobile applications (in short as apps) has become unprecedentedly important. To find failures (e.g., crashes and display issues) in apps before their shipment to end users, a practical way is to conduct automated User Interface (UI) testing, where testing intends to mimic human interactions with mobile devices. By employing automated UI testing, developers can conveniently test different versions of their apps in batch on multiple devices for a long period of time. This practice complements manual UI testing, which is usually needed for complex functionalities but can be expensive and prone to mistakes.

Although numerous automated UI testing tools [3, 4, 8, 19, 27, 28, 30, 36, 55, 58] for mobile apps have been developed, they often do not satisfy two major industrial requirements that make a tool desirable in industrial settings. The first requirement is high applicability across platforms, where one can apply the tool without modifications or with only small modifications across devices of various platforms. While Android and iOS are the dominant mobile operating systems globally, a great number of mobile operating systems have increasingly emerged to split the market. For example, Microsoft has released Surface Duo [33] as one of its major mobile platforms. Some countries have developed mobile platforms [17, 24, 32] to avoid relying on overseas technologies. The growing trend of diversified mobile platforms makes a unified testing solution increasingly desirable for the industry. The second requirement is supporting non-standard UI elements in

apps, where the internal structure of on-screen contents cannot be obtained using platform APIs (e.g., UIAutomator [23] from the Android framework). Non-standard UI elements are usually embedded web UI [44] or customized UI elements. For many apps such as Taobao [18], embedded web UI elements provide a convenient way for online and prompt deployment of new features without modifying the app client code or bothering users to update the app. Mobile games [51] also heavily utilize customized UI elements from graphic engines, which maintain their own internal UI structures. Figure 1 shows one instance of non-standard UI elements in Taobao. Many sophisticated UI testing tools [8, 13, 19, 20, 42] do not work properly in such cases given that these tools cannot identify UI elements from on-screen contents through obtained on-screen content structures (namely *UI hierarchies*).

To help satisfy the preceding industrial requirements, visual testing tools [48, 52] emerge to leverage Computer Vision (CV) techniques to identify UI elements from *only* screenshots, bringing two major benefits. First, a visual testing tool tends to be non-intrusive to the target device and is generally applicable to any platform, saving considerable development and maintenance costs. Tool developers and quality assurance engineers need to maintain and learn the usages of only a single tool. Second, a visual testing tool complements automated testing tools that rely on UI hierarchies obtained using platform APIs. Visual testing tools are capable of handling non-standard UI elements, enabling more functionalities to be explored and improving the testing effectiveness.

Given the usefulness of visual testing in industry practices, in this paper, we report our experiences of developing and deploying VTest, our industrial automated visual testing framework, for testing the Taobao app in Alibaba. Taobao is a highly popular e-commerce app in China, with about **one billion** monthly active users. In 2020, sales in Taobao reached **498.2 billion Chinese Yuan** in a single day, and the peak value of orders reached 583,000 transactions per second.

VTest addresses two major tasks, *UI-Element Identification* and *Test-Action Planning*, along with providing the infrastructure support for automated visual testing. UI-Element Identification aims to identify UI elements from raw screenshot images to provide necessary information (e.g., boundaries) of UI elements for subsequent steps in the testing process. Test-Action Planning adopts a specialized algorithm that decides on the action to perform (e.g., tapping a specific location on the screen) on the test device based on the identified UI elements from the current screen (and optionally, previous screens).

To support the preceding two major VTest tasks, we build a hardware infrastructure with two major hardware components, inspired by the progress of robotic testing [37]. In particular, we employ a high-speed camera to capture screenshots for UI-element identification and a robotic arm to execute actions fired by test-action planning. The hardware infrastructure provides full applicability across platforms, i.e., the infrastructure is applicable across various platforms without modification.

Considering the monetary costs of the hardware infrastructure, we also build a software infrastructure based on only lightweight platform APIs [15, 23, 45] from Android and iOS to support the preceding two major VTest tasks. In particular, the software infrastructure invokes the screen capturing platform API to capture screenshots for UI-element identification, and invokes the action execution platform API to execute actions fired by test-action planning. Compared to the hardware infrastructure, the software infrastructure incurs lower monetary costs and provides lower (but sufficiently high) applicability across platforms.

To instantiate VTest for building a practical solution in industry settings, we report our experiences of selecting proper techniques to address the two preceding tasks, respectively. For the task of UI-element identification, we examine the effectiveness of existing state-of-the-art identification techniques [7, 14, 38] on representative scenarios from Taobao. Our experiences show that there is no one-size-fit-all solution to deal with various scenarios, and a context-sensitive solution is desirable for UI-element identification in practice. For the task of test-action planning, we design a random strategy. Our evaluation results show that, even instantiated with the simple random strategy, VTest outperforms Monkey [16] with 87.6% more activity coverage.

We have deployed VTest both internally in Alibaba and externally in the Software Green Alliance to provide services for top smartphone vendors and app vendors in China. Internally, in Alibaba, VTest is used for everyday testing of a dozen popular industrial apps, such as Taobao. VTest is also used to test parts of Alipay, a highly popular payment app with over **one billion** users. Specifically, we conduct user experience testing based on visual testing with hardware infrastructure and conduct other testing tasks based on visual testing with software infrastructure. Externally, we have also started providing VTest as a public testing service in the Software Green Alliance (*SGA* in short) [1]. SGA is formed by top smartphone vendors (e.g., Huawei) and app vendors (e.g., Baidu, Netease, and Tencent) to collaboratively assure the quality of popular industrial apps on massive devices. The service is improving quality assurance for hundreds of popular industrial apps.

In summary, this paper makes the following main contributions:

- Raising awareness of the importance of visual testing for industry practice and academic research.
- A practical automated visual testing framework, VTest instantiated with carefully designed techniques and hardware/software infrastructure support.
- Experiences and lessons learned from developing and deploying VTest in Alibaba and the Software Green Alliance.

2 BACKGROUND ON NON-STANDARD ANDROID UI ELEMENTS

We use the term of *non-standard UI elements* for UI elements whose internal structures or contents cannot be obtained using the Android accessibility APIs. These UI elements are usually represented by Java classes outside the `android.widget` package. Most existing testing tools rely on UIAutomator [23] (provided by the Android framework) to capture on-screen contents, where UIAutomator works by invoking the accessibility APIs upon each UI element on the screen. Consequently, these tools are unable to know about the inner contents of non-standard elements. The tools need inner contents to learn about actionable regions (e.g., where can be clicked) or useful attributes (e.g., texts) inside these UI elements. UI state abstraction, if conducted by a testing tool, also becomes infeasible

Figure 1: An example of non-standard UI elements compared with standard UI elements on Android.

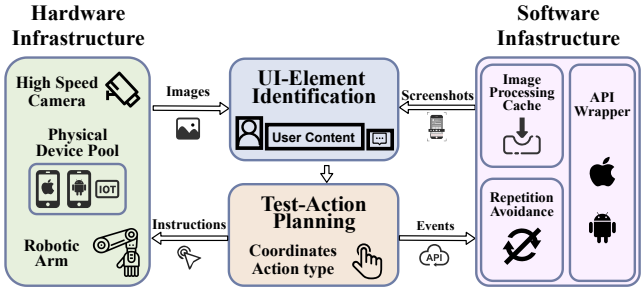

Figure 2: Overview of the VTest framework.

in the cases where non-standard elements make up most of the UI elements from the screen under analysis.

Figure 1 illustrates an example of non-standard UI elements on Android. Figures 1-I(b) and 1-II(b) show the simplified UI hierarchies captured by UIAutomator corresponding to the screenshots in Figures 1-I(a) and 1-II(a). Red boxes in Figures 1-I(a) and 1-II(a) indicate the boundaries of UI elements (represented as nodes) shown in Figures 1-I(b) and 1-II(b), respectively.

The Taobao app in Figure 1-I(a) uses WebView, an embedded web browser module provided by Android. UIAutomator is incapable to access the inner contents of WebView elements. Thus, only the outer boundary of the WebView element is included in the UI hierarchy as shown in Figure 1-I(b). On the contrary, another app in Figure 1-II(a) uses standard Android UI elements, and the detailed UI contents can be captured as shown in Figure 1-II(b).

3 INFRASTRUCTURE SUPPORT FOR VISUAL TESTING

Automated visual testing satisfies industrial needs from two major aspects. First, automated visual testing identifies UI elements from *only* screenshots, making visual testing highly applicable without modifications or with only small modifications across devices of various platforms. Second, automated visual testing can identify the internal structures of non-standard UI elements (see Section 2 for an example), while platform APIs may not be able to provide such information. To support automated visual testing, we build both the hardware infrastructure and software infrastructure, which have respective different applicable scopes and deployment expenses.

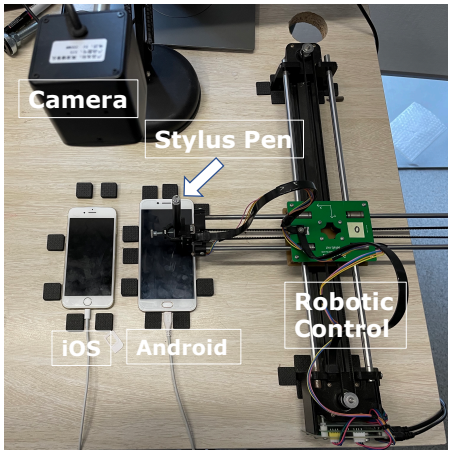

Figure 3: A scene of our hardware infrastructure in action.

3.1 Hardware Infrastructure with Robotic Arms and Cameras

We adopt a robotic testing scheme, using high-speed cameras and robotic arms to build the supporting hardware infrastructure for automated visual testing. Figure 3 shows a scene of our hardware infrastructure in action.

Screenshot Capturing. A high-speed camera captures, processes (e.g., recognizes the screenshots from raw images), and transmits the screenshots for subsequent UI-element identification. While we adjust the positions of the camera and the mobile device under test to make sure that the whole screen of the device is inside the scope of the captured image, the surrounding region also appears in the image. In practice, we place a white/black board under the device to reduce the interference brought by the background and use the OpenCV library [6] to segment devices' screens from the background.

Action Projection. VTest analyzes the screenshot and determines the test action to be taken next. The action is represented as an action type along with coordinates (e.g., click on (100,200)). To enable the action execution on physical devices, we project (i.e., transform) the action from the coordinate system in the screenshot to the coordinate system in the real world. We adopt a (simplified) camera calibration [57] technique and projection-based transformation [41] to convert actions fired by VTest into real physical actions on devices. As shown in Figure 3, we place small black blocks closely around the edge of the device and record their positions in the real world. After we properly pose the camera vertical to the device screen, we capture a screenshot and use the OpenCV library to locate the black blocks in the screenshot. With the coordinates of the same object in the screenshot and the real world, we obtain a matrix describing the one-to-one transformation from coordinates in the screenshots to the real-world coordinates.

Action Execution. Consisting of a robotic control module (an XY plotter [25]) and a stylus pen, the robotic arm interacts with device screens physically. Upon receiving the coordinates, the robotic

control module first drives the stylus pen to the corresponding position. The module then lowers the pen to contact the screen for a specific amount of time and lifts the pen off the screen. We set the contact time period based on the type of the target action, e.g., 50 millseconds to simulate a single tap and 200 milliseconds for a long click. The robotic arm additionally supports other types of actions such as swiping (by contacting the screen, moving to a given direction for a specific distance, and leaving the screen).

3.2 Software Infrastructure with Low Costs

While the hardware infrastructure is promising for its general applicability to *any* platform, its hardware development and maintenance costs are high for deployment. To provide visual testing with relatively low costs and with sufficient applicability across platforms, we build a software infrastructure using only lightweight platform APIs from UIAutomator [23] and Web Drive Agent (WDA) [15] to be applicable on the Android and iOS platforms. In particular, the software infrastructure relies only on the screen capturing and action execution platform APIs. With only small modification on the invocation of the two kinds of lightweight platform APIs, the software infrastructure can be applicable across platforms. The software infrastructure invokes the screen capturing platform API to capture screenshots for UI-element identification, and invokes the action execution platform API to execute actions fired by test-action planning, supporting automated visual testing with low costs and sufficient applicability.

4 UI-ELEMENT IDENTIFICATION WITH COMPUTER VISION

To instantiate VTest, we first identify UI elements from screenshots. By regarding UI-element identification from screenshots as a domain-specific object detection task, we can leverage various object detection techniques including both deep-learning-based [14, 21, 38, 39] and old-fashioned [7, 34, 35, 43] computer-vision techniques to achieve this goal. Inspired by an empirical study [7] conducted on the Rico dataset [11], we conduct an empirical study on the effectiveness of existing UI-element identification techniques for Taobao. The results show that existing different techniques achieve the best effectiveness in different scenarios. We dive into the principles of the techniques to explain their effectiveness, and design an integrated technique to achieve the best effectiveness.

4.1 Study Setup

Object Detection Techniques. We select five representative object detection techniques to evaluate their effectiveness for Taobao. Three of the techniques are deep-learning-based while the other two are old-fashioned, covering the mainstream techniques of object detection.

- **Faster-RCNN** [39] is a two-stage anchor-based deep-learning-based technique for object detection. It first uses a region proposal network to extract regions of interest (called anchor boxes or bounding boxes) that possibly contain objects, and then classifies the inside objects with another neural network.
- **YOLOv3** [38] is a one-stage anchor-based deep-learning-based technique for object detection. Different from Faster-RCNN, YOLOv3 generates anchor boxes and classifies the inside objects at the same time.
- **CenterNet** [14] is a one-stage anchor-free technique for object detection. Instead of using anchor boxes, it detects objects with the positions of their keypoints including corner and center points.
- **Xianyu** [53] is an old-fashioned computer vision technique developed by Alibaba to reverse-engineer GUIs. We use its element detection part only, where Xianyu binarizes, slices, and detects the edges of UI elements.
- **UIED** [7] is an old-fashioned computer vision technique enhanced by deep-learning-based Optical Character Recognition (OCR). It detects non-textual UI elements with old-fashioned techniques and textual UI elements with EAST [59].

For Faster-RCNN, YOLOv3, and CenterNet, we use their pre-trained models [7]. For UIED, we replace its OCR model with our own.

UI Screen Collection in Taobao. To comprehensively examine the effectiveness of the preceding object detection techniques, we select five representative scenarios in Taobao to evaluate their effectiveness. In each scenario we manually collect 12 distinct screenshots.

- **Messenger.** The messenger scenario is one of the main functionalities in Taobao, where users communicate with each other.
- **Goods detail.** The goods detail scenario demonstrates the goods with images and texts, and is the most common scenario in Taobao.
- **Live streaming.** The live streaming scenario is an increasingly popular way for users to do online entertainment, where the widgets are mixed with their background.
- **Shop Front.** The shop front scenario displays goods in a flow, and is the major scenario for merchants to promote their products.
- **Game.** The game scenario is another complicated scenario where the widgets are mixed with their background.

Effectiveness Metrics. We evaluate the correctness of the positions of the bounding boxes detected for the given screenshot s against the set of ground-truth bounding boxes on s. For each detected bounding box b, we calculate the Intersection over Union (in short as IoU, referring to the intersection area of the two boxes over the their union area) with each of the ground-truth bounding boxes on s, and find b's matched ground-truth bounding box as the one that has the largest IoU with b and this IoU is higher than the predefined threshold. When we successfully find b's matched ground-truth bounding box, we determine b as a True Positive (TP). Otherwise, we determine b as a False Positive (FP). We determine ground-truth boxes that are not matched with any detected bounding box as False Negatives (FNs). We use *Precision*, *Recall*, and *F1-score* to measure the effectiveness of the preceding techniques. *Precision = TP/(TP+FP)* measures the fraction of detected bounding boxes that really contain UI elements. *Recall = TP/(TP+FN)* measures the capability of detecting as many UI elements as possible in the screenshots. F1-score = $2 \times Precision \times Recall / (Precision + Recall)$ represents the identification accuracy comprehensively.

We manually collect the screenshots in the preceding scenarios and use UIAutomator to obtain the bounding boxes of clickable UI

Table 1: Effectiveness of various UI-element identification techniques on different scenarios.

Techniques	CenterNet			YOLOv3			Faster-RCNN			Xianyu			UIED		
Scenario	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**
Messenger-**0.5**	0.347	0.311	0.328	0.917	0.098	0.177	0.517	0.282	0.365	0.184	0.114	0.141	0.603	0.532	**0.565**
Messenger-**0.75**	0.328	0.294	0.309	0.866	0.090	0.164	0.496	0.274	0.353	0.116	0.074	0.090	0.600	0.529	**0.562**
Messenger-**0.95**	0.308	0.278	0.292	0.021	0.003	0.006	0.050	0.030	0.038	0.001	0.001	0.001	0.600	0.529	**0.562**
Goods-**0.5**	0.341	0.328	0.334	0.675	0.072	0.130	0.527	0.351	0.422	0.266	0.167	0.205	0.748	0.306	**0.434**
Goods-**0.75**	0.339	0.327	0.333	0.636	0.067	0.122	0.459	0.305	0.367	0.241	0.150	0.185	0.738	0.301	**0.428**
Goods-**0.95**	0.321	0.310	0.315	0	0	0	0.004	0.003	0.003	0.029	0.020	0.024	0.677	0.278	**0.394**
Live-**0.5**	0.263	0.380	0.311	0.675	0.072	0.130	0.480	0.458	**0.469**	0.314	0.105	0.158	0.540	0.219	0.311
Live-**0.75**	0.257	0.371	0.304	0.662	0.087	0.154	0.430	0.413	**0.422**	0.249	0.085	0.127	0.512	0.216	0.304
Live-**0.95**	0.245	0.355	0.291	0	0	0	0.007	0.007	0.007	0.024	0.012	0.016	0.508	0.213	**0.301**
Shop-**0.5**	0.328	0.433	0.373	0.834	0.132	0.228	0.500	0.415	0.454	0.214	0.145	0.173	0.541	0.540	**0.541**
Shop-**0.75**	0.308	0.405	0.350	0.679	0.105	0.182	0.486	0.404	0.441	0.158	0.108	0.128	0.539	0.538	**0.538**
Shop-**0.95**	0.289	0.380	0.328	0.025	0.004	0.007	0.057	0.047	0.051	0.007	0.004	0.005	0.537	0.536	**0.537**
Game-**0.5**	0.281	0.429	0.340	0.6	0.092	0.159	0.490	0.513	**0.501**	0.168	0.106	0.130	0.572	0.214	0.311
Game-**0.75**	0.269	0.413	0.326	0.516	0.079	0.137	0.467	0.494	**0.480**	0.120	0.073	0.091	0.572	0.214	0.311
Game-**0.95**	0.238	0.363	0.287	0	0	0	0.011	0.013	0.012	0.007	0.003	0.004	0.572	0.214	**0.311**
Average-0.5	0.312	0.376	0.337	0.740	0.093	0.165	0.503	0.404	**0.442**	0.229	0.127	0.161	0.601	0.362	0.432
Average-0.75	0.300	0.362	0.324	0.672	0.086	0.152	0.468	0.378	0.413	0.177	0.098	0.124	0.592	0.360	**0.429**
Average-0.95	0.280	0.337	0.303	0.009	0.001	0.003	0.026	0.020	0.022	0.014	0.008	0.010	0.579	0.354	**0.421**

Notes: **-0.5**, **-0.75**, and **-0.95** represent the IoU threshold being set to 0.5, 0.75, and 0.95, respectively. **Prec.** refers to the *precision* metric. **F1** refers to the *F1-score* metric (identification accuracy).

elements, serving as the ground truth of UI-element identification. In addition, we manually include the bounding boxes of non-native UI elements that are clickable. All screenshots are resized to 1080 × 540 (50% of the original size) for efficiency.

We conduct UI-element identification using the selected techniques on the selected scenarios, respectively.

4.2 Empirical Results and Analysis

In this section, we first present the empirical results, analyze the reasons behind the results, and then propose an integrated UI-element identification technique using the insights gained from the analysis.

4.2.1 UI-Element Identification Accuracy. Table 1 shows the results of different object detection techniques on different scenarios, from which we have five observations.

Comparison among deep learning techniques. Among the three selected deep-learning-based models, Faster-RCNN achieves the highest average identification accuracy when the IOU threshold is set to 0.5. CenterNet follows Faster-RCNN w.r.t. average identification accuracy. Generally, the two-stage technique (Faster-RCNN) achieves better effectiveness than one-stage techniques due to its explicit stage of proposing bounding boxes. However, when setting the IOU threshold to 0.95, the identification accuracy of Faster-RCNN and YOLOv3 drops drastically, while the accuracy of CenterNet only slightly drops. This result is due to the working principle of the three models. Faster-RCNN and YOLOv3 are anchor-based techniques, both of which require predefined scale and aspect ratio of bounding boxes, i.e., these techniques assume the fixed size and shape of detected bounding boxes. Since the size and shape of UI elements vary in a large range, the anchor-based techniques cannot precisely identify UI elements. In contrast, CenterNet adopts an anchor-free detection schema to enable its precise UI-element identification capability.

Generalization ability of deep learning techniques. When compared to their original effectiveness in previous work [7], the effectiveness of the three models all drop substantially despite being trained with over 66,000 screenshots from the Rico dataset [11]. YOLOv3 achieves the worst effectiveness among the three deep-learning-based techniques. To validate its effectiveness given enough training data, we collect 1000 UI screens in Taobao as the training dataset to fine-tune the YOLOv3 model. We use *mean Average Precision* (*mAP*) to evaluate the effectiveness of YOLOv3 models. The mAP metric is similar with F1-score. The mAP of the original YOLOv3 trained on the Rico dataset is 0.02, while the mAP of the YOLOv3 trained on our collected dataset is 0.75. The results show that the variance of UI elements accounts for the effectiveness drop of deep-learning-based techniques.

This result reflects that the high variance of UI elements makes it hard for deep-learning-based techniques to generalize even trained on a large dataset. Even in a single app such as Taobao, the UI elements can change drastically across versions. If deep-learning-based techniques fail to generalize, each time when the app is updated, a new set of training data is required, raising additional maintenance cost and limiting the application of deep-learning-based techniques of UI-element identification in industry.

Comparison among old-fashioned techniques. For old-fashioned techniques, UIED substantially outperforms Xianyu. UIED, specially designed for UI-element identification, achieves the state-of-the-art identification accuracy due to its insight of separately detecting text and non-text elements and involving a deep-learning-based OCR model. Xianyu, another old-fashioned technique, fails to deal with complicated UI screen patterns in real-world industrial apps.

Comparison between deep-learning-based and old-fashioned techniques. UIED achieves the best effectiveness in most scenarios. On average, deep-learning-based techniques outperform Xianyu but are much worse than UIED. Since old-fashioned techniques do not require training data, UIED is promising to be deployed as the

Table 2: Classification of UI elements and their corresponding identification techniques.

Bound \ Inner.	Rich	Simple
Clear	old-fashioned	old-fashioned
Mixed	deep-learning-based	None

Notes: ***Bound*** *refers to the boundary of a UI element and its background.* ***Inner.*** *refers to the internal structure of a UI element.*

backbone UI-element identification technique for automated visual testing for Taobao.

Effectiveness divergence in different scenarios. However, in the live streaming scenario, UIED performs poorly and deep-learning-based techniques outperform old-fashioned techniques. We inspect these scenarios and find out that in these scenarios, UI elements are mixed with the background images, which are impossible for old-fashioned techniques to identify. We further inspect other scenarios and similar situations also exist.

4.3 In-depth Analysis of the Effectiveness Divergence

Based on our preceding empirical results, we find that the characteristics of UI elements make UI-element identification distinguished from generic object detection. From a detection perspective, we categorize a UI element from two aspects: (1) is there a clear boundary between the UI element and the background? (2) is the internal structure of the UI element of high variety? The answers to the preceding two questions divide UI elements into four classes, depicted in Table 2.

If a UI element has a clear boundary with its background, then the UI element can be easily identified by old-fashioned techniques. If the UI element is constant (i.e., have little change of its internal structure), then deep-learning-based techniques can easily memorize it in the training set, and generalize easily to only background changes.

Inspired by our preceding empirical findings, we combine the deep-learning-based and old-fashioned techniques in two steps to obtain better identification accuracy with relatively low costs. First, we run YOLOv3 and UIED in parallel to get two sets of identified UI elements. Second, for each UI element u_i identified by YOLOv3, we search in the results from UIED. If u_i is completely inside a UI element from UIED or u_i has no intersection with any UI element from UIED, we add the UI element into the final set of identified UI elements. Third, we remove a UI element identified by UIED if it is too large (which we regard as a detection failure for UI elements mixed with the background). Finally, we get the UI elements identified by the integration of deep-learning-based and old-fashioned techniques. We choose YOLOv3 out of the three deep-learning-based techniques because of its high precision, avoiding too many false positives to burden the second step (merging the results from two techniques). We choose UIED out of the two old-fashioned techniques because of its dominating effectiveness over Xianyu.

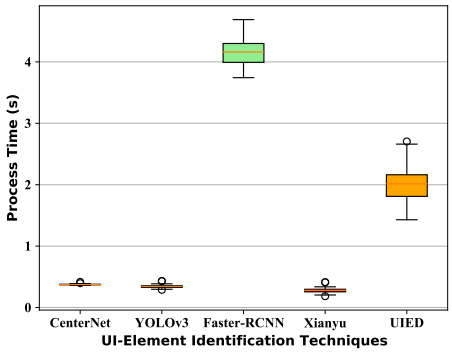

Figure 4: Statistics of processing time for a single screenshot of each technique.

5 EFFECTIVENESS OF VTEST WITH UI-ELEMENT IDENTIFICATION

5.1 Test-action Planning Strategy

Although there exist sophisticated techniques for test-action planning [5, 19, 27, 30, 36, 42, 58], we find it difficult to straightforwardly adapt them for two main reasons. First, some techniques require information beyond on-screen contents, such as code coverage [5, 30, 42] and the internal IDs of UI elements. The requirement breaks our assumption of using only screenshots as a source of information. Second, for techniques that require knowing only on-screen contents, they heavily rely on precise information (e.g., matching exact boundaries of UI elements) about the on-screen contents to conduct state abstraction. While these techniques can reliably obtain precise UI information through platform APIs, the outputs of UI-element identification techniques are often prone to errors and noises—even state-of-the-art techniques of UI-element identification achieve only up to 60% detection accuracy as shown in Table 1.

Consequently, we decide that at the current stage, a randomized exploration strategy is adopted as the test-action planning strategy for VTest. At each round, VTest randomly selects one UI element from the identified UI elements and *clicks* on it. To simulate system events, before each round, with probability $\alpha = 0.005$, VTest restarts Taobao; with probability $\beta = 0.08$, VTest performs a swipe action toward a random direction; with probability $\gamma = 0.08$, VTest goes back to the previous UI screen.

5.2 Infrastructure Support to Boost Efficiency of VTest

In addition to the exploration strategy, we find that the image processing can be time-consuming, taking seconds to process a single screenshot. Figure 4 shows the processing time of each technique. The set of screenshots is the same with those used in Section 4.

Faster-RCNN is the most time-consuming due to its two-stage technique, taking 4.2 seconds on average to process a single screenshot. CenterNet, YOLOv3, and Xianyu are time-efficient, taking no more than 0.5 seconds to process a single screenshot. YOLOv3 and Xianyu are extremely fast, taking only 0.2 seconds on average to process a single screenshot. UIED takes the second longest time, about 2.2 seconds on average, to process a single screenshot.

Table 3: Activity coverage and execution speed.

Baselines	Activity Coverage	Execution Rounds
Cache-Test	**16.5%(+ 87.6%)**	11,753
Comb-Test	15.1%	9,382
U-Test	14.8%	9,477
F-Test	12.2%	5,267
Y-Test	11.6%	17,493
X-Test	9.1%	15,925
C-Test	6.7%	17,003
Monkey	8.8%	108,000

To improve the efficiency and reduce the cost brought by image processing, we cache the historical processing results, and use perceptual hashing [26] to match screenshots. Every time when VTest captures a new screenshot, we calculate its hash distance with the historical screenshots. If the hash distance is larger than $\tau = 0.9$, then we regard the new screenshot as the same with the cached result. If a screenshot is cached, we reuse the identification results instead of invoking a new UI-element identification process.

5.3 Evaluations of VTest for Taobao

Baseline Techniques. We compare the effectiveness of VTest with different UI-element identification techniques. We denote VTest instantiated with YOLOv3, Faster-RCNN, Xianyu, CenterNet, and UIED as **Y-Test**, **F-Test**, **X-Test**, **C-Test**, and **U-Test**, respectively. We denote VTest instantiated with the combination of YOLOv3 and UIED as **Comb-Test**. To examine the effectiveness of the infrastructure support, we denote VTest instantiated with UIED and the cache strategy as **Cache-Test**. We also compare VTest with Monkey, a state-of-the-practice tool. For Monkey, we use its default settings, and set the throttle to 200 millseconds.

Environmental Settings. Since we use random testing as our exploration strategy, we run each tool continuously for 6 hours to reduce the random influence. To further compensate for the potential influence brought by randomness, we run each test (a single run lasting for 6 hours) for 3 times. We use the same tester account for the same baseline techniques, and re-login the app before each run to keep the testing environment as same as possible. We run the tools on a Huawei P30 phone with Android 10 to collect activity coverage information. Taobao has over 500 activities, and its functionalities are rigorously organized according to activities, each group of which is maintained by a group of engineers. Consequently, activity coverage is a good metric to examine the effectiveness of automated testing tools for Taobao. We run the experiments on a Ubuntu 20.04 machine with an Intel Core i7-8700 CPU and an Nvidia Tesla V100 GPU. We run a daemon process that uses Android Debug Bridge (ADB) to collect the name of the app activity that the testing tool is exploring. If the activity name does not belong to the targeted app, the daemon process automatically restarts the targeted app to keep the exploration inside it.

5.4 Experimental Results

Table 3 shows the activity coverage achieved by VTest instantiated with different techniques. We also list the number of execution rounds (i.e., total actions taken) in 6 hours. U-test outperforms Monkey with 68.2% relative improvement. The number of Monkey's execution rounds is approximately counted by dividing the 6-hour test time with the throttle instead of counting the number of low-level events that Monkey sends to the device. VTest fires 90% fewer actions than Monkey does, indicating that widget-awareness can drastically improve the effectiveness of automated testing for Taobao.

Y-Test and F-Test outperform Monkey with 31.8% and 38.6% relative improvement, respectively. Although Faster-RCNN has higher UI-element identification accuracy than YOLOv3, it takes much more time to process screenshots than YOLOv3, causing to achieve similar activity coverage. These results show that it is important to take the processing speed of UI-element identification techniques into consideration when we apply these techniques to automated visual testing.

Despite its fast processing speed, Xianyu is not desirable for automated visual testing due to its extremely low detection accuracy. The poor effectiveness of C-Test indicates that identification accuracy drawn from a not-end-to-end empirical study does not assure good effectiveness for automated visual testing.

Combining deep-learning-based techniques and old-fashioned techniques does not improve the activity coverage while the combination can improve the identification accuracy. The UI elements missed by UIED may not contribute to the improvement of the activity coverage, and this result does not overthrow our conclusion of using combined techniques.

Finally, the cache strategy improves the effectiveness with an additional 19.4% activity coverage (87.6% in total), highlighting the importance of infrastructure improvement in addition to algorithm innovations.

6 LESSONS LEARNED FROM DEVELOPING AND DEPLOYING VTEST

6.1 Deployment Experiences

Currently, we have deployed VTest both internally in Alibaba and externally in the Software Green Alliance to provide testing services for a large group of partners, including major mobile phone manufacturers and app vendors in China.

6.1.1 ***Internal Deployment.*** We deploy VTest in Alibaba for the quality assurance of many highly popular industrial apps such as Taobao, Tmall, Taobao Live, Qianniu, Fliggy, Xianyu, Youku, and DingTalk. In addition, in Ant, VTest is used for testing parts of Alipay. Most of these apps each have over **100 million** active users. For example, Alipay[1] is the most popular mobile payment app in China with over **one billion** active users, while DingTalk[2] is a highly popular messenger app with over **500 million** active users across **19 million** organizations to communicate for teamwork.

VTest provides two kinds of visual testing for internal usages. First, leveraging the hardware infrastructure, VTest can precisely test the runtime performance of the app under test on given devices, especially those low-end/inexpensive devices. Running the software infrastructure on these devices burdens their operating systems (OSs), taking seconds or even failing to capture screenshots or execute events with the software infrastructure. In addition, the

[1]https://intl.alipay.com/
[2]https://www.dingtalk.com/en

software infrastructure overpasses the physical touch on devices and executes events directly using APIs. In contrast, the hardware infrastructure simulates the real touch like humans and can detect issues found in actual usages, such as a click-response failure with a screen protector pasted on the device screen. Being fully non-intrusive, the hardware infrastructure can precisely monitor the loading time and reaction time to reflect users' authentic experiences when using the app. Second, VTest with the software infrastructure provides automated visual testing for the daily quality assurance (such as functionality testing) of iOS and Android devices at a low cost.

6.1.2 ***External Deployment.*** We provide automated visual testing as a public testing service for the Software Green Alliance (SGA) [1]. SGA is a business alliance in China to coordinate top vendors of mobile devices and apps to assure the quality of popular industrial apps on popular devices (i.e., assuring the compatibility of multiple apps on multiple devices). The members of SGA consist of Huawei, Alibaba, Baidu, Tencent, and many other large IT and Internet companies in China. The mobile device vendors in the alliance provide a variety of devices including the latest and recent device models, and the mobile app vendors test their recent apps on the devices to ensure the compatibility of their apps on the devices. This practice requires cross-platform applicability since the diversified models of devices can be equipped with a modified version of existing OSs or a completely new OS.

To provide a large-scale service, we deploy VTest in a cloud-edge fashion: we put UI-element identification and test-action planning modules in Alibaba Cloud and devices under test in a physical computer room. We use physical devices instead of emulators since developing high-performance emulators is expensive and the behaviors on emulators may not truly reflect those on physical devices. For the hardware infrastructure, due to the cost factor, currently, we hold only one computer room to place the robotic arms and cameras. We plan to increase the number of robotic arms and cameras and seek better hardware equipment with lower cost and higher efficiency than the current ones (XY Plotters).

6.2 Lessons Learned from Developing VTest

We next summarize five major lessons learned during our development and deployment of VTest.

6.2.1 ***We should make the best shot with the best gun.*** We expect desirable matching between problems and solutions. For visual testing, the problem space includes testing against high-end and low-end devices, along with functionality testing and user experience testing with different concerns and requirements. The solution space includes the hardware infrastructure and software infrastructure, with different levels of costs and benefits. We should avoid the mismatch of problems and solutions, and pursue the exact match of problems and solutions. For example, it is not cost-effective to leverage the hardware infrastructure for functionality testing, bringing in additional monetary and runtime costs. For user experience testing, on the contrary, using the software infrastructure may not effectively discover failures that users may encounter during app usage. When an action is executed (using the software infrastructure) too fast compared to the regular usage by actual users, the detected failures are false positives and hence not that important. When testing on low-end/inexpensive devices, the runtime overhead of the software infrastructure is relatively high, and only the hardware infrastructure can adequately handle the testing tasks on these devices.

In addition, we should balance the benefits and costs brought by different techniques. We should avoid a one-size-for-all solution and carefully determine the applicable scope for each technique.

Monetary costs matter. The hardware infrastructure is an exciting solution to provide general applicability to any platform and is dispensable for the user experience testing given its full non-intrusiveness. However, in practice, it is expensive to adopt the hardware infrastructure for large-scale usage (e.g., deploying hundreds of robotic arms and high-speed cameras). To reduce hardware costs, when testing multiple devices simultaneously (e.g., compatibility testing across platforms), we use only one set of robotic arms and execute actions on each device one by one, avoiding purchasing multiple sets of robotic equipment. We also develop the software infrastructure for the consideration of monetary cost. Mobile platforms have platform APIs available for screen capturing and event execution, of which we make the best use to develop the software infrastructure supporting automated visual testing. The software infrastructure requires almost no equipment cost and retains high cross-platform applicability for functionality testing.

Runtime costs matter. In addition to monetary costs, the runtime costs during the testing are also not negligible. When testing on high-end devices, the hardware infrastructure is too slow to efficiently explore the functionalities of an app, while the software infrastructure suffers when testing on low-end/inexpensive devices. We should carefully select a proper infrastructure to minimize the runtime costs for efficiency and effectiveness.

Amortizing costs helps. While automated visual testing inevitably incurs costs from image processing, by reusing the processed results for other tasks, we can amortize the costs among multiple tasks. For example, we store all the processed results produced during the automated visual testing and later use them for detecting visual failures such as image rendering failures, image loading failures, and violations of UI design principles (being our ongoing work).

6.2.2 ***Modular tool development enables valuable reuse opportunities.*** To provide a concrete way of cost amortization, we should be open-minded and enable the opportunities of reusing modules of a tool for other tasks. VTest is originally designed for automated visual testing and its modules are reused for three other tasks. First, the hardware infrastructure is used to execute test scripts for automating manual testing of user experience. Second, the hardware infrastructure is also used for testing the interaction between multiple devices testing cross-platform compatibility. The two preceding reuses amortize costs of developing and deploying the hardware infrastructure. Third, our ongoing effort is applying the designed techniques of UI-element identification to review the mini-programs (for detecting visual failures and violations of UI design principles) in Taobao before their shipment to users, amortizing the image processing costs.

6.2.3 ***There is a need to investigate individual cases.*** Conventionally, people conduct empirical studies of various techniques

and select the best one according to the average effectiveness. But the best technique may not necessarily dominate other techniques in all cases, and the average accuracy cannot reflect that some techniques perform the best in certain cases while other techniques perform the best in other cases. For example, similar to a finding in a previous study [7], our empirical study shows that UIED, an old-fashioned technique, achieves the highest average identification accuracy. However, based on the investigation of individual cases, we observe that neither old-fashioned techniques nor deep-learning-based techniques dominate each other; in contrast, they complement each other in different cases. Such phenomenon suggests a combination of preceding complementary techniques. Despite the existence of multiple techniques, their combination still faces challenges. To address these challenges, investigation of individual cases is important to find the characteristics of different techniques and their advantages for specific cases. Based on the investigation, we design a combination mechanism of deep-learning-based and old-fashioned techniques depicted in Section 4.

We should pursue the possibility to combine multiple techniques instead of selecting the single best one according to the average numbers, and to properly combine multiple techniques, we should investigate individual cases in two aspects. First, we should check the existence of complementarity among different techniques with the investigation of individual cases. Second, if the complementarity exists, rules need to be derived to determine under what conditions a certain technique should be adopted over other techniques.

6.2.4 ***There is a need to embrace non-deep-learning-based techniques and their integration.*** The preceding lesson learned can be revisited here when multiple techniques under combination include deep-learning-based techniques, which often can demonstrate the best average effectiveness in recent studies [54]. But non-deep-learning-based techniques can also achieve good effectiveness, sometimes even surpassing deep-learning-based techniques as shown in Table 1. Despite the easy-of-use nature of deep-learning-based techniques in the inference phase, the training phase can be expensive for industry deployment, requiring months to collect tens of thousands of training data [56]. Our experiences show that if apps evolve and the UI changes, an additional dataset is needed, increasing the maintenance costs. In addition, even if we cannot solve the problem entirely with non-deep-learning-based techniques, we can divide the problems into pieces and solve them one by one. We can use non-deep-learning-based techniques to solve a group of problem pieces, making deep-learning-based techniques more desirable in terms of cost and effectiveness for the remaining problem pieces.

6.2.5 ***Improving infrastructure support's efficiency matters besides algorithm improvement.*** In addition to algorithm innovation, we should pay attention to infrastructure support. Wang et al. [45] point out the importance of infrastructure for Android testing, and we draw similar conclusions for automated visual testing. For example, image processing is relatively expensive (in terms of the runtime and monetary costs) and our infrastructure stores the processing results for each screenshot. Before processing a new screenshot, we first look up the cache using the perceptual hashing match algorithm. If we detect a hash matching (the edit distance is less than a threshold), the infrastructure returns the cached results instead of conducting image processing (running the techniques of UI-element identification) again. The caching mechanism can generally improve the effectiveness and reduce the costs of any type of automated visual testing. Take VTest with UIED as an example: by caching the historical results, the infrastructure gains 19.4% additional activity coverage improvement and reduces approximately 20% invocations of image processing. Since the infrastructure support is orthogonal to algorithm innovation, we should pay attention to both of them and simultaneously improve them to obtain good results.

7 RELATED WORK

Automated UI testing for mobile apps. There have been numerous techniques proposed to achieve good testing effectiveness (e.g., high code coverage) through automated UI testing upon mobile apps, predominantly on the Android platform. The history traces back to Monkey [16], developed by the Android team and shipped with every Android device. The tool simply samples and executes actions from a predefined probability distribution of action categories without acquiring any information about the app UI. However, the tool is extremely efficient and is shown to achieve good testing effectiveness, even outperforming numerous research tools [9, 46]. Consequently, Monkey is still widely used in the industry. Other sophisticated tools use various strategies for test-action planning, e.g., building and querying a UI transition model [8, 13, 19, 20, 42], systematically exploring UIs [4, 5, 29], and relying on randomized evolution [28, 30, 50].

Visual testing. There exists a rich literature of leveraging computer vision techniques in software testing [2, 10, 21, 22, 49, 52]. Sikuli [52] and JAutomate [2] leverage visual record-and-replay techniques to enable testers to write visual test scripts that use images instead of textual descriptions, benefiting the test automation. Choudhary et al. [10], He et al. [21, 22] use computer vision techniques to test cross-browser compatibility. White et al. [48] leverage a deep-learning model to detect widgets from images to facilitate automated testing on open-source Java-based desktop applications.

Robotic testing. Instead of simulating a user's GUI actions via internal OSs or GUI framework, robotic testing [12, 31, 37, 47] leverages robotic arms to simulate GUI actions externally. Most robotic testing frameworks and tools focus on automating test execution. Dhanapal et al. [12] use robotic arms to test hardware functionalities and performance of smart devices. Qian et al. [37] leverage a capture-and-replay technique to generate visual test scripts from videos and replay these scripts on IoT devices.

8 CONCLUSION

In this paper, we have reported our experiences from developing and deploying VTest, an automated visual testing framework in an industry setting, aiming to mainly address the industrial requirements for applicability across platforms and capability to handle non-standard UI elements. We have developed integrated techniques and the hardware/software infrastructure support to instantiate VTest, outperforming state-of-the-practice tool Monkey with 87.6% coverage improvement. We have deployed VTest both internally in Alibaba and externally in the Software Green Alliance to

provide testing services for hundreds of highly popular industrial apps. We have additionally summarized five major lessons learned from development and large-scale deployment of VTest.

ACKNOWLEDGMENTS

Tao Xie's work was partially supported by National Natural Science Foundation of China (Grant No. 62161146003), a grant from Alibaba, and XPLORER PRIZE.

REFERENCES

[1] 2021. Software Green Alliance. https://www.china-sga.com/sga/resource/cloudtest.html
[2] Emil Alegroth, Michel Nass, and Helena H Olsson. 2013. JAutomate: A Tool for System- and Acceptance-test Automation. In *ICST*.
[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *ASE*.
[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *FSE*.
[5] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *OOPSLA*.
[6] Gary Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer* (2000).
[7] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object Detection for Graphical User Interface: Old Fashioned or Deep Learning or a Combination?. In *ESEC/FSE*.
[8] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *OOPSLA*.
[9] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *ASE*.
[10] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. Webdiff: Automated Identification of Cross-Browser Issues in Web Applications. In *ICSM*.
[11] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *UIST*.
[12] Karthikeyan Balaji Dhanapal, K Sai Deepak, Saurabh Sharma, Sagar Prakash Joglekar, Aditya Narang, Aditya Vashistha, Paras Salunkhe, Harikrishna GN Rai, Arun Agrahara Somasundara, and Sanjoy Paul. 2012. An Innovative System for Remote and Automated Testing of Mobile Phone Applications. In *SRII*.
[13] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-Travel Testing of Android Apps. In *ICSE*.
[14] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. 2019. Centernet: Keypoint Triplets for Object Detection. In *ICCV*.
[15] Facebook. 2018. WebDriverAgent. https://github.com/facebookarchive/WebDriverAgent
[16] Google. 2021. Android Monkey. https://developer.android.com/studio/test/monkey
[17] Alibaba Group. 2021. AliOS. https://aliosthings.io/#/
[18] Alibaba Group. 2021. Taobao. https://taobao.com
[19] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications via Model Abstraction and Refinement. In *ICSE*.
[20] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *MobiSys*.
[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *ICCV*.
[22] Meimei He, Guoquan Wu, Hongyin Tang, Wei Chen, Jun Wei, Hua Zhong, and Tao Huang. 2016. X-Check: A Novel Cross-Browser Testing Service Based on Record/Replay. In *ICWS*.
[23] Xiaocong He. 2018. Python wrapper of Android uiautomator test tool. https://github.com/xiaocong/uiautomator
[24] Huawei. 2020. Harmony OS. https://www.harmonyos.com/en/
[25] RR Jegan, E Gnanasundaram, M Gowtham, R Sivanesan, and D Thiyagarajan. 2018. Modern Design and Implementation of XY Plotter. In *ICICCT*.
[26] Evan Klinger. 2013. Perceptual Hashing. http://www.phash.org/
[27] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *ASE*.
[28] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *ESEC/FSE*.
[29] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *FSE*.
[30] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *ISSTA*.
[31] Ke Mao, Mark Harman, and Yue Jia. 2017. Robotic Testing of Mobile Apps for Truly Black-Box Automation. *IEEE Software* (2017).
[32] Ltd. Meiwang Technology Co. 2021. Flyme. https://www.flyme.cn/
[33] Microsoft. 2020. Surface Duo. https://www.microsoft.com/en-us/surface/devices/surface-duo?activetab=overview
[34] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *TSE* (2020).
[35] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *ASE*.
[36] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *ISSTA*.
[37] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. RoScript: A Visual Script Driven Truly Non-Intrusive Robotic Testing System for Touch Screen Applications. In *ICSE*.
[38] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *CVPR*.
[39] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection With Region Proposal Networks. In *NeurIPS*.
[40] StatCounter. 2021. Mobile Market Share Worldwide. https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide
[41] Gilbert Strang, Gilbert Strang, Gilbert Strang, and Gilbert Strang. 1993. *Introduction to linear algebra*. Wellesley-Cambridge Press Wellesley, MA.
[42] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *ESEC/FSE*.
[43] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J Ko. 2018. Rewire: Interface Design Assistance from Examples. In *CHI*.
[44] Android Development Team. 2011. Webviewclient hooks list. http://developer.android.com/reference/android/webkit/WebViewClient.html
[45] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools. In *ISSTA*.
[46] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *ASE*.
[47] Ragnar Wernersson. 2015. *Robot Control and Computer Vision for Automated Test System on Touch Display Products*. Master's thesis.
[48] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving Random GUI Testing with Image-Based Widget Detection. In *ISSTA*.
[49] Zhen Xu and James Miller. 2018. Cross-Browser Differences Detection Based on an Empirical Metric For Web Page Visual Similarity. *TIT* (2018).
[50] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *MoMM*.
[51] Jiaming Ye, Ke Chen, Xiaofei Xie, Lei Ma, Ruochen Huang, Yingfeng Chen, Yinxing Xue, and Jianjun Zhao. 2021. An Empirical Study of GUI Widget Detection for Industrial Mobile Games. In *FSE*.
[52] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *UIST*.
[53] Chen Yongxin, Zhang Tonghui, and Chen Jie. 2019. UI2code. https://laptrinhx.com/ui2code-how-to-fine-tune-background-andforeground-analysis-2293652041/
[54] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, and Qianxiang Wang. 2022. Automated Assertion Generation via Information Retrieval and Its Integration with Deep Learning. In *ICSE*.
[55] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated Test Input Generation for Android: Are We Really There yet in an Industrial Case?. In *FSE*.
[56] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *CHI*.
[57] Zhengyou Zhang. 2000. A Flexible New Technique for Camera Calibration. *TPAMI* (2000).
[58] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android:Towards Getting There in an Industrial Case. In *ICSE-SEIP*.
[59] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. 2017. East: An Efficient and Accurate Scene Text Detector. In *CVPR*.