RoScript: A Visual Script Driven Truly Non-Intrusive Robotic Testing System for Touch Screen Applications

Ju Qian
MIIT Key Laboratory of Safety-Critical Software, Nanjing University of Aeronautics and Astronautics
Nanjing, China
jqian@nuaa.edu.cn

Zhengyu Shang
Shuoyan Yan
Yan Wang
Nanjing University of Aeronautics and Astronautics
Nanjing, China

Lin Chen
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
lchen@nju.edu.cn

ABSTRACT

Existing intrusive test automation techniques for touch screen applications (e.g., Appium and Sikuli) are difficult to work on many closed or uncommon systems, such as a GoPro. Being non-intrusive can largely extend the application scope of the test automation techniques. To this end, this paper presents RoScript, a truly non-intrusive test-script-driven robotic testing system for test automation of touch screen applications. RoScript leverages visual test scripts to express GUI actions on a touch screen application and uses a physical robot to drive automated test execution. To reduce the test script creation cost, a non-intrusive computer vision based technique is also introduced in RoScript to automatically record touch screen actions into test scripts from videos of human actions on the device under test. RoScript is applicable to touch screen applications running on almost arbitrary platforms, whatever the underlying operating systems or GUI frameworks are. We conducted experiments applying it to automate the testing of 21 touch screen applications on 6 different devices. The results show that RoScript is highly usable. In the experiments, it successfully automated 104 test scenarios containing over 650 different GUI actions on the subject applications. RoScript accurately performed GUI actions on over 90% of the test script executions and accurately recorded about 85% of human screen click actions into test code.

CCS CONCEPTS

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

KEYWORDS

test automation, non-intrusive, robot, GUI testing, computer vision

ACM Reference Format:
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 *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380431

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380431

1 INTRODUCTION

Touch screen applications, including the ones on smartphones and tablets and the ones on embedded devices, are increasingly used in entertainment, education, industry, etc. To ensure the quality of these applications, careful testing should be conducted [13, 21, 27, 29, 35, 43, 44]. However, the testing can be tedious and costly. Without proper tools, a tester may need to manually and repeatedly operate on a screen, which can be labor-intensive.

Automating the testing of touch screen applications can largely increase test efficiency and reduce test cost. One way for test automation is to record manual actions as test scripts and use test tools to automatically execute test scripts as many times as desired. Typical scripting and test automation frameworks include textual script based ones like UIAutomator [8], Robotium [6], and Appium [1] which use labels, indexes, coordinates, etc. to identify widgets and express GUI actions, and visual script based ones like Sikuli [16], JAutomate [12], and EggPlant [3] which leverage widget images and computer vision techniques for visual GUI testing [11, 15].

All the above test automation techniques are intrusive (to the devices under test). They require supports from the underlying operating system (OS) or GUI framework of an application under test (AUT) to get GUI states and trigger GUI actions. If the AUT is running on a closed system with no underlying OS or GUI framework support accessible (e.g., a GoPro camera or a Nintendo Switch game console) or an uncommon platform whose underlying system support is hard to be accessed (e.g., a customized VxWorks-like embedded system), then such test automation may not be able to be conducted. For some AUTs, even though the underlying systems might be accessible, hacking into these systems to enable test automation may change the tested environments to some extent (e.g., change the performance and energy consumption), and the GUI actions triggered via internal system facilities may not closely emulate the experience of real users. Under such circumstances, the test results of an AUT may not be trustable.

There need non-intrusive test automation techniques for touch screen applications with no necessary underlying system support easily accessible or allowed to be accessed. A possible way for non-intrusive testing is using cameras and physical robots to conduct testing [5, 18, 26, 33, 42]. Among the existing work, the techniques in [26, 33] seem to have the ability of automating the testing of touch screen applications with robots. However, they use exact

coordinates to drive test execution. Manually determining the exact coordinates of a GUI widget is too costly, and using fixed coordinates for test automation also lacks robustness [10]. The techniques in [26, 33] require hacking into the subject mobile devices in a pre-test-execution step to gather detailed GUI action coordinates and therefore are not truly non-intrusive regarding the whole test cycle.

A more usable and truly non-intrusive solution to the test automation problem would be using visual test scripts [3, 12, 16] to drive robot-based test execution. Visual scripts are expressive, robust to changes in spatial arrangements of GUI widgets, and independent of an AUT's underlying GUI framework [15, 16]. They can be non-intrusively created and thus are very suitable for non-intrusive testing. However, there is yet no such test automation framework or system in the literature. Many challenges are also faced in developing a visual script driven robotic testing approach.

First, how to express various robot-based test actions on a touch screen in visual test scripts? The existing visual script languages do not support robot controlling and cannot be used for convenient and accurate text inputting in robotic testing. Extensions to such languages should be made to fit the robotic testing context.

Second, how to execute visual test scripts with a robot? The existing visual GUI test engines do not support robotic testing, and the existing robotic test engines cannot execute visual test scripts. A new test engine must be designed to carefully plan robot motions according to visual test scripts and camera photos of a tested device to ensure fully automated and highly accurate test execution.

Third, how to automatically record GUI actions on a touch screen into test scripts in order to reduce the test script creation cost? All the existing test script recording techniques are intrusive. They cannot work in a truly non-intrusive context. New non-intrusive script recording techniques need to be introduced.

This work addresses the above challenges and introduces RoScript, a truly non-intrusive and comprehensively evaluated robotic visual GUI testing system for touch screen applications. RoScript uses an extended visual test script language to express and drive test actions on a touch screen. Different from traditional visual script languages, the new test script language uses non-intrusively obtained camera-taken pictures, instead of images accurately captured from a GUI screen via an AUT's underlying OS, for widget localization and GUI state verification. It also provides special text input and robot control instructions to support robotic testing.

RoScript performs GUI actions by an XY plotter, a robot originally used for personal writing and drawing and cheap enough to be adopted in daily testing. It integrates a computer vision based test engine to turn actions expressed in visual scripts into low-level robot commands. We introduce several techniques like screenshot cropping, screenshot-to-real-world move translation, keyboard model based key pressing, and robot detouring to enable fully automated and highly accurate test execution (Section 6). RoScript does not depend on any OS or GUI framework to get GUI states or trigger GUI actions. Its test automation is completely non-intrusive and hence applicable to touch screen applications running on almost arbitrary platforms whatever their underlying OSs or GUI frameworks are. Such generally applicable test automation is especially valuable when intrusive testing is unavailable (e.g., when doing third party testing for beta or off-the-shelf products with no internal test support facilities exported) or too costly (e.g., when intrusive testing tools need to be newly designed for platforms unsupported by the existing tools).

To reduce the test script creation cost, we also propose a computer vision based technique to automatically record touch screen click actions into test scripts from videos of human actions on a device under test (DUT). The script recording is based on visually recognizing human hand actions on the screen. Different from the existing script recording techniques, it is fully non-intrusive and therefore fits well for non-intrusive test automation.

We conducted experiments on 21 applications running on 6 different touch screen devices, including Android/iOS smartphones, a Windows tablet, a Raspberry Pi, and a GoPro to validate the effectiveness of the proposed approach. The results show that RoScript successfully automated 104 test scenarios containing over 650 different GUI actions on the subject applications. It accurately performed GUI actions on 91.4% of the test script executions, without triggering the actions in unexpected ways. 84.5% of the touch screen click actions can be correctly recorded into test scripts from human actions in the application usage videos. These results suggest that RoScript is highly usable for practical touch screen application testing.

The main contributions of this work are:

(1) A truly non-intrusive robotic test automation system. The system integrates a new test device, an extended visual test script language, and a computer vision based test engine to enable truly non-intrusive test automation.

(2) A video-based script recording technique. We introduce a truly non-intrusive computer vision based technique which can automatically create test scripts from camera-recorded videos of human hand actions on a touch screen.

(3) A comprehensive evaluation of the proposed robotic testing technique. The evaluation on the flexibility, accuracy, and efficiency of the new test automation technique shows it is practical for use.

The rest of this paper is organized as follows. Section 2 shows a motivating example for the approach. Section 3 discusses the related work. Section 4 - 7 introduce the detailed approach. Section 8 presents the evaluation. Finally, we conclude the paper and discuss future work in Section 9.

2 MOTIVATING EXAMPLE

Fig. 1 shows a test scenario for the application embedded in a GoPro device. The scenario tests the video deletion feature of the GoPro device. There are 6 steps to delete a recorded video: (1) swipe the screen to enter the control menu; (2) press button  to go to the playback view; (3) click image to select a video; (4) click widgets  and  to start and confirm deletion; (5) press button  to return to the playback view; and (6) check the existence of  to determine the successfulness of the deletion.

Figure 1: A GoPro video deletion test scenario

To automate the testing of the GoPro video deletion feature, traditional intrusive testing tools cannot be used. This is because an off-the-shelf GoPro device is a closed system with no underlying system facilities accessible and it runs an uncommon OS. As far as we know, none of the existing tools like Appium or Sikuli supports automating the testing on such a device. The robotic testing techniques in [26, 33] are also unusable, because it is extremely difficult to hack into the GoPro device to get the GUI action coordinates and manually determining the coordinates is too costly. There needs a non-intrusive and easy to use test automation technique for GoPro-like devices.

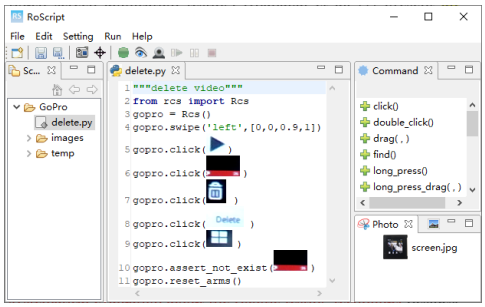


Figure 2: A test script for the GoPro video deletion feature

Our RoScript is the first robotic testing system which can meet the above GoPro test automation requirement. It uses visual test scripts composed by instructions and widget images to express the GUI actions on the GoPro touch screen (see Fig. 2). Different from the exact widget coordinates, widget images can be easily cropped from camera photos of the GoPro touch screen. By using visual scripts to drive an external robot to conduct testing, the testing can be non-intrusively automated. During the non-intrusive testing, no additional software needs to be installed on the GoPro, and no cable or wireless connection needs to be established between the GoPro and the RoScript test tool.

3 RELATED WORK

3.1 Expressing GUI Actions

The GUI actions on a touch screen can be expressed via textual or visual test scripts. The textual scripts relying on screen coordinates lack robustness [10]. The textual scripts using labels, indexes, etc. to identify widgets usually depend on GUI frameworks to locate widgets on a screen [1, 6, 8] and thus are difficult to be used in non-intrusive testing. Compared with them, visual scripts largely use images to express GUI actions [3, 12, 16] and rely on computer vision techniques rather than GUI framework supports for test automation. They are better choices for non-intrusive testing.

We use visual test scripts for robotic testing. To fit the robotic testing context, we make two extensions to the existing visual script languages [3, 12, 16]. First, we use camera-taken pictures to identify GUI widgets. Second, we introduce new instructions for text inputting and robot controlling. The camera-taken pictures may have low quality, and the camera-photo-driven robot-based testing is very different from traditional visual GUI testing [3, 12, 16] which just needs to match same-scale images on accurate screenshots for test execution. For these reasons, special image processing techniques like screenshot cropping and scale normalization are also introduced, and a new script execution engine is designed to execute the extended visual test scripts.

3.2 Simulating GUI Actions

Traditional test automation frameworks [1, 3, 6, 8, 12, 16] simulate a user's GUI actions via internal OSs, GUI frameworks, or VNC-like [9] remote desktop facilities [11]. These test automation approaches are intrusive. When the internal OSs, GUI frameworks, or VNC services are inaccessible, intrusive testing cannot be conducted.

A possible direction for non-intrusive testing is simulating GUI actions externally via robots [5, 18, 26, 33, 42]. However, the approaches in [5, 18, 42] focus on using robots to test hardware functionalities and performance of smart devices or to do remote testing. They do not support automatically simulating GUI actions on various widgets on a touch screen and therefore are not generally usable for test automation of touch screen applications.

Kanstrén et al. [26] and Mao et al. [33] present two robotic approaches for automating the testing of touch screen applications. They support non-intrusive test execution. However, Kanstrén et al.'s approach requires installing special software on the subject mobile devices to collect detailed device usage data before testing. Mao et al.'s approach needs to firstly do intrusive testing with other tools on the subject devices to obtain low-level event sequences [32]. Both of them require hacking into mobile devices in a pre-test-execution step to gather information like action types and coordinates. Regarding the whole testing cycle, they are not truly non-intrusive.

We also use robots to simulate GUI actions for test automation. Our work differs from the existing work in the followings. First, we leverage visual test scripts to drive robotic testing truly non-intrusive regarding the whole test preparation and execution cycle. Second, we introduce an intelligent robotic visual GUI testing engine armed with many techniques to enable fully automated and highly accurate test execution. The engine relies on images and computer vision techniques to achieve test automation, while the existing work [26, 33] depends on exact coordinates for automated test execution. Third, most of the existing work uses specially designed robots or industrial robots for testing, while we use a commodity personal XY plotter for test automation, and we have addressed various problems faced in building a practical visual GUI testing system with such a kind of robots. Fourth, we present the first experimental evaluation about the robotic test automation techniques, while neither [26] nor [33] is with experimental results presented.

In the literature, there is also some work testing the robots themselves [14, 17, 25, 31, 34, 36]. It aims at problems quite different from ours.

3.3 Recording Test Scripts

We non-intrusively record test scripts from human action videos. This is very different from the traditional script recording techniques. Almost all the existing test tools, e.g., [1, 6, 8, 12, 16, 19–24, 28, 39], demand information like action types, coordinates, and images obtained via OSs or GUI frameworks to record test scripts. Such a kind of script recording is intrusive. Luan et al. [30] also

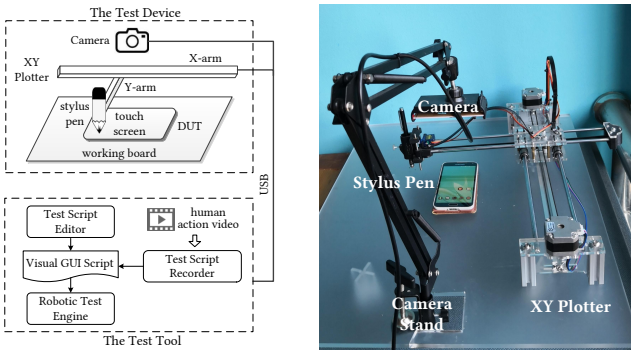
record test scripts from videos. However, they obtain application usage videos by installing a specially designed screen-capturing tool on the DUTs. Such video obtaining is intrusive, and their script generation is based on information embedded into the videos by their screen-capturing tool. Our approach differs from the existing video-based script recording in two ways. First, we use camera-taken videos recording human actions on a touch screen as the basis for script generation. The video recording is truly non-intrusive without needing to install any special software on the DUTs. Second, we generate test scripts by analyzing human action videos with computer vision techniques. Besides the videos, no additional information is required for test script generation.

4 OVERVIEW OF THE ROBOTIC TESTING SYSTEM

The architecture of the RoScript robotic testing system is illustrated in Fig. 3(a). Touch screen devices are put onto a working board for testing. No special connection is required to be established on these devices. We use an XY plotter to conduct touch screen actions. The X- and Y-arms of the XY plotter control the action positions, and a stylus pen in the XY plotter is responsible for performing screen touches. We use a camera to obtain the screen states of a touch screen application. The camera and the XY plotter are connected to the RoScript test tool (Fig. 2) running on a personal computer.

In the test tool, touch screen actions are expressed in visual GUI scripts. A robotic test engine turns high-level GUI actions into low-level robot control commands so that the XY plotter can move as expected. We provide a test script recorder, in addition to a test script editor, to help create visual GUI scripts.

Fig. 3(b) shows the robot used for testing. The arms of the XY plotter are driven by stepper motors with a moving accuracy of 0.2mm. The working area of the XY plotter is about A4 paper size. The camera is set to a fixed position via the camera stand during testing. In total, the hardware of the test device costs about 200 USD, which is cheap enough for daily testing.



(a) system architecture (b) the test device

Figure 3: The RoScript test automation system

Under the robotic testing system, the testing of a touch screen application can be conducted with the following steps. The first step is to create test scripts for the AUT. In addition to manual script creation, RoScript integrates script recording techniques to help automatically generate test code. To record a test script, a tester needs to manually use the subject touch screen application for one round with all human actions recorded into a camera video. Then, test code for the click actions will be automatically generated from the video, and the screen states between different actions will also be extracted to support manual test script modification.

The second step is to put the robot and the DUT at proper positions and set necessary configurations. We require the DUT to be put in parallel to the X-arm of the test robot. The camera position needs to be adjusted to ensure that the whole DUT is visible by the test tool. We also need to move the stylus pen of the test robot to the top left corner of the DUT, so that the initial position of the pen can be easily calculated. The only configurations that need to be manually set before testing are the width, height, and thickness of the touch screen device. RoScript is, to some extent, robust to small errors in these adjustments and configurations. It calculates the touch positions and touch depths based on such settings. In most cases, a slight error in a calculated touch position or a calculated touch depth does not affect triggering a GUI action from its widget area of a certain size via an elastic tip in the stylus pen.

In the third step, we execute the created test scripts with the robotic test engine. Assertions in a test script are used to verify the test results.

5 THE VISUAL TEST SCRIPT LANGUAGE

We use a Python-based language to support robot-based visual GUI testing. Some key instructions supported by RoScript are listed in Table 1. The GUI action triggering instructions like `click`, `drag`, and `swipe` are commonly used in many visual GUI testing tools. The result verification instructions like `assert_exist` and `assert_not_exist` are borrowed from Sikuli [16]. In these instructions, like Sikuli, the `image` parameter can also be replaced by a string, and an optional `region` parameter specifying a restricted area for matching images on the screen is also supported.

Besides the commonly used instructions, we introduce instructions `take_screen_photo`, `reset_arms`, `move`, `move_outside_of_screen`, etc. for test robot controlling. Instruction `take_screen_photo` takes a camera photo of the DUT, crops the screen area out, and returns it as the result. Instruction `reset_arms` resets the robot arms to their initial positions. The `move` instruction moves the robot's stylus pen to a relative position. The `move_outside_of_screen` instruction moves the robot arms outside of the screen area so that they will not affect taking full camera photo of the screen.

Table 1: Some key test instructions provided by RoScript

#	Instruction	Description
1	click(*image, region*): bool	Click a specified image on the screen.
2	drag(*image*1, *image*2, *region*1, *region*2): bool	Drag an object from a position specified by image1 to a position specified by image2.
3	swipe(*direction, region*)	Swipe on the touch screen.
4	assert_exist(*image, region*): bool	Assert the existence of an image.
5	assert_not_exist(*image, region*): bool	Assert the non-existence of an image.
6	take_screen_photo(): image	Take a photo of the DUT's touch screen.
7	reset_arms()	Reset the robot arm position.
8	move(*dx, dy*)	Move the stylus pen to a position (*dx, dy*) relative to the current point.
9	move_outside_of_screen()	Move robot arms outside of the screen area.
10	press_keyboard(*model, keys*)	Press a sequence of keys on a soft keyboard specified by the given model.

When doing text inputting, for non-intrusive testing, it is impossible to call OS facilities to simulate key pressing. One choice is using instructions like `click(a)` to click key buttons in a soft keyboard on a touch screen in order to input characters. However, in that way, to input a long string, we may need to crop a lot of key images and add a lot of click actions into the test script. This can be tedious. Another way is using an instruction like `click`("abc") to input a string, depending on OCR (Optical Character Recognition) techniques to locate the key buttons corresponding to the characters 'a', 'b', and 'c'. However, OCR techniques are often without enough accuracy on low-quality camera-taken photos, and a character like 'a' may occur at many places other than the soft keyboard on the screen. Such a way frequently fails. To address the above limitations, we introduce a keyboard model based instruction `press_keyboard`(*model*, *keys*) for text inputting. The instruction takes a specified keyboard model and a sequence of key names for inputting. It is convenient for use and depends on the specified model to enable accurate inputting. The implementation of this instruction will be introduced in Section 6.4.

With the provided test instructions, RoScript can express various activities on a touch screen with test scripts.

6 THE ROBOTIC TEST ENGINE

A robotic test engine is responsible for controlling the robot to perform test actions. The engine leverages computer vision algorithms to turn the GUI actions expressed in a test script into low-level robot move commands [2].

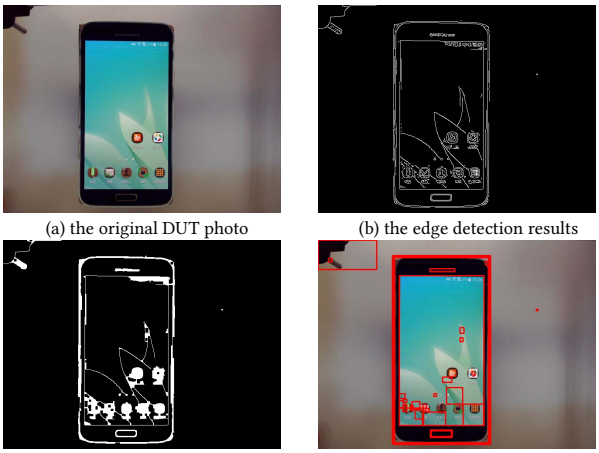
6.1 Locating Widgets on the Screen

To perform a test action with a robot, we take a photo of the current touch screen and then locate the target widget's image specified in the test script on the screen photo to determine the action position. There can be various resolutions for taking camera photos. High resolutions can lead to better widget localization accuracy, while low resolutions result in faster widget localization speed. We use a resolution of 1600x1200 with a good balance between widget localization accuracy and speed for taking photos.

The widget localization is via computer vision algorithms. It takes two steps. The first step crops a screenshot from a camera-taken photo of the DUT. The second step matches images in a test script on the cropped screenshot to locate widgets.

6.1.1 Cropping Screenshots. Besides the touch screen, robot arms, as well as objects like a power cable or a piece of paper on the working board, may be taken into camera photos of the device under test. These objects can affect the localization accuracy and speed of widgets on a screen. To avoid their negative effects, it is necessary to crop the screen area (screenshot) out of a camera-taken DUT photo and focus just on the screen area for better widget localization.

We propose a computer vision algorithm to find the touch screen area in a camera photo for screenshot cropping. The algorithm includes five key steps. The first step uses Canny edge detection [41] to detect the edges in the camera photo. The second step performs a morphological closing operation [41] on the edge detection results to connect the broken edges as much as possible. The third step detects all the contours from the edges obtained in the second step and finds the rectangle regions of these contours. The detected contour regions may be separated or overlapped. They are the candidates for screen area recognition. Fig. 4 shows an example demonstrating the results of these three steps, where the finally detected contour regions are marked with red border rectangles in Fig. 4 (d).



(a) the original DUT photo (b) the edge detection results

(c) after doing closing operation (d) the detected contour regions

Figure 4: Crop the screen area from a photo of the DUT

The fourth step filters out the contour regions with too large sizes. This is because in our DUT photos like that in Fig. 4 (a), a screen area must be with enough size difference from the original photo. The upper bound used for filtering is: $size(camera\_photo) \times U_{dev}$, where $U_{dev} = 0.95$.

In the fifth step, we find the largest contour region among the ones retained by the last step. This region is recognized as the touch screen area of the DUT. In Fig. 4 (d), the rectangle with bold red borders marks the screen area identified from Fig. 4 (a).

With the above algorithm, a touch screen area including the borders of the DUT will be obtained. We can run the algorithm again on a border-including touch screen image to get the exact screen area. However, the identification of the border-excluding touch screen area is easy to be affected by the contents on the screen. Therefore, RoScript only runs the algorithm once to get border-including touch screen images. Such a screenshot cropping is enough for our robotic testing.

6.1.2 Widget Localization. We locate widgets on a screenshot with a template matching algorithm. There are many such algorithms in the literature [40]. We tested the built-in template matching algorithms in OpenCV [4] and the recent BBS algorithm [38] on screen photos. According to the results, RoScript picks the normalized correlation coefficient matching algorithm in OpenCV which performs best in both matching accuracy and speed for implementation.

There can be a scale difference between a widget image in a test script and the image of the same widget obtained at test execution time. Such a scale difference may affect the template matching accuracy and efficiency. To address the scale difference, we store the sizes of the screenshots containing the in-test-script widget images. When doing template matching for a widget, the sizes of the DUT screenshot taken at runtime and the screenshot of the

in-test-script widget image are compared. If there is a difference, we automatically normalize the scales to the same level according to these sizes in advance of template matching. In that way, the effects of a scale difference on widget matching can be largely reduced.

6.2 Determining the Move of Robot Arms

Having located the target widget of a GUI action on a screenshot, it is straightforward to determine the move of the touch position between GUI actions on the screenshot space. However, a move on the screenshots is different from a move of stylus pen in the real world. We need to translate an on-screenshot move to a real-world move.

To do such translation, firstly a robot coordinate system (RCS) for controlling the robot moves in the real world and a screenshot coordinate system (SCS) expressing the positions of objects on the screenshot space are defined, as demonstrated in Fig. 5. A move on the SCS can be denoted as a position change $\langle \Delta x_s, \Delta y_s \rangle$. The key problem in determining robot move is to translate $\langle \Delta x_s, \Delta y_s \rangle$ to a position change on the robot coordinate system, i.e., $\langle \Delta x_r, \Delta y_r \rangle$.

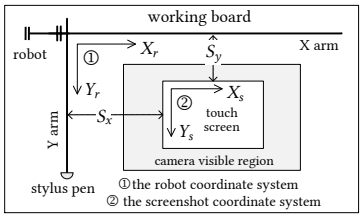


Figure 5: The coordinate systems

There can be a scale difference between an object in the real world (RCS) and the same object on a screenshot (SCS). For coordinate translation, we need to know the real size of the cropped screenshot area (DUT size) before testing, with which the scale rate ρ_{sr} between SCS and RCS can be determined. Let the origin of the SCS be at point $\langle S_x, S_y \rangle$ on the RCS. For a point $\langle x_s, y_s \rangle$ on the screen coordinate system, its coordinates on the robot coordinate system are:

$$x_r = x_s/\rho_{sr} + S_x, y_r = y_s/\rho_{sr} + S_y.$$

According to the above relationships, the translation from $\langle \Delta x_s, \Delta y_s \rangle$ to $\langle \Delta x_r, \Delta y_r \rangle$ is:

$$\Delta x_r = \Delta x_s/\rho_{sr}, \Delta y_r = \Delta y_s/\rho_{sr},$$

with which the move of robot arms can be determined.

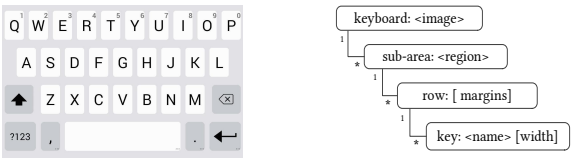
6.3 Performing GUI Actions

RoScript performs a GUI action by executing a sequence of robot motions. The below presents the ways to perform two key basic actions. More complex actions can be performed in similar ways.

- *Click.* A click action consists of the following steps: (a) move the stylus pen on the robot coordinate system to the click position; (b) move the pen down by a calculated depth; (c) move the pen up. The pen moving down depth is calculated by subtracting the thickness of the tested device from a predefined depth, i.e.,
$$depth_{click} = depth_{predefined} - thickness(DUT).$$
- *Swipe.* We perform swipe actions on a specified region, the default of which is the full screen. Take swiping up for example, i.e., `swipe('up')`. The first step is moving the stylus pen to a position near the bottom center of the region. Then, we move the pen down to touch the screen. Under the touching state, the pen will be moved for a length toward the top of the region. After that, we move the pen up, and a swipe action completes.

6.4 Pressing Keyboards

RoScript introduces a keyboard model based approach for text inputting. The approach defines a model for each soft keyboard to express the distributions of keys on the screen. A typical keyboard model for the 26-key keyboard is demonstrated in Fig. 6. The model contains a keyboard image for locating the keyboard area on the screen. The keys inside the keyboard are organized in a sub-area, row, and key hierarchy. Each sub-area has its relative region information. Rows and keys are in-default assumed to be evenly spread. Non-default size rows and keys have their relative margin or width information. With such information, it is enough to calculate the center coordinates of each key by providing minimal information in the keyboard model.



(a) keyboard image (b) model hierarchy

Figure 6: A keyboard model

When inputting a character like `e` via a 26-key keyboard with instruction `press_keyboard`('26-key', 'e'), we first locate the keyboard area by matching the corresponding keyboard image on the screen. Then, inside the located keyboard area, we use the keyboard model to calculate the screen coordinates of key `e`. With such coordinates, character `e` can be inputted via a keypress.

The approach uses the whole keyboard area, instead of each individual key button, for image matching. This can ensure the matching accuracy and hence suffer less from the possibly low quality of camera-taken screenshots. It can handle the cases when there are multiple occurrences of a character on the screen. The approach also does not require cropping key images for text inputting, which will ease the creation of test scripts.

6.5 Connecting Actions

To connect different touch screen actions, RoScript automatically makes a detour, moving the robot arms outside of a certain region after performing a GUI action to ensure these arms will not affect the camera photo taking of the next action. The move is determined by the image matching region used by the next test instruction. If the next instruction is a `move`, a `swipe`, etc. doing no image matching, then no such move is actually needed. Otherwise, an image matching region R used by the next instruction will be found. The region can be the whole screen or a region determined by the arguments of the next instruction. There are three candidate directions to move outside of region R, illustrated as A, B, and C in Fig. 7. Among A, B, and C, RoScript chooses a point with the shortest moving distance to move the robot arms outside of the image matching region.

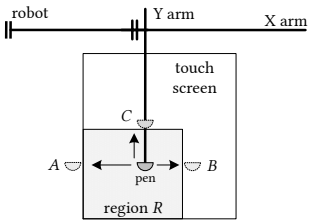
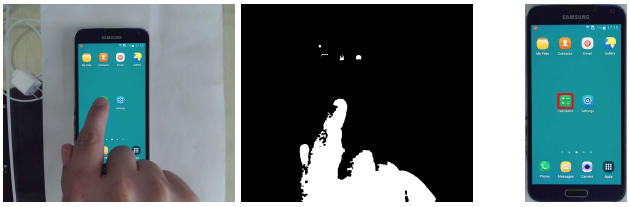


Figure 7: Moving the stylus pen outside of a region

7 RECORDING TEST SCRIPTS FROM VIDEOS

RoScript can automatically record human click actions on a touch screen into test scripts from the corresponding videos. Non-click actions like drag and swipe are difficult to recognize. For such actions, only semi-automatic script recording is supported.

Before recording, the subject touch screen device needs to be put onto a board, with a camera hanging over it focusing on the screen. Then, we use the camera to record human actions on the touch screen into videos. For accurate script recording, when operating on the screen, we require the user to use only a single fingertip to touch the screen, and the other non-touching fingers should be in a gripping state (Fig. 8(a)). To avoid the perspective effect as much as possible, when doing clicking, the angle between the screen and the touching finger should better be less than 45°. After each click, we require the user to move her hand completely out of the camera photo region.



(a) human screen touch (b) binary image after skin detection (c) widget image cropping

Figure 8: The automatic script recording

On a human action video, RoScript introduces a five steps algorithm to record the human actions into a test script.

Step 1: *Extract image frames and detect fingers.*

In this step, RoScript extracts image frames from the recorded video. For each extracted frame, a color-range-based skin detection [37] is used to detect the hand area in the frame. The detection result is turned into a binary image, an example of which is shown in Fig. 8(b). We then detect the contour of the hand region in the binary image. To reduce the effects of noises, only the largest contour close to a border of the binary image is regarded as the hand contour. This is because when a user touches the screen, her hand always crosses a border of the camera photo region. If no such contour is found, there is considered no finger in the frame.

For the detected hand contour, we regard the top point in the contour as the fingertip position. Let the bottom left corner of the frame be with coordinates $\langle 0, 0\rangle$ and the X and Y axes coordinates of a point increase when moving the point up to the top right corner. Then, the fingertip position in a frame is $\langle x_f, y_f\rangle$ with a maximal y_f value in all points of the hand contour. If there is considered no finger in the frame, the fingertip position is set to $\langle 0, 0\rangle$.

Step 2: *Group frames by actions.*

In the next step, we group frames by actions according to the vertical position of the fingertip in each frame. Fig. 9 shows the changes of the fingertip vertical position in a sequence of frames extracted from an example human action video. Here vertical position 0 means no finger in a frame. A click action corresponds to a sequence of frames with the fingertip vertical position starting from 0, gradually increased to a peak point, and finally decreased to 0 again. According to this finding, for the example in Fig. 9, it is easy to group the frames into the ones corresponding to 5 actions.

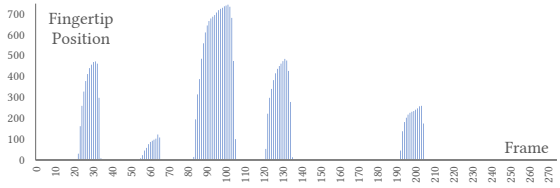


Figure 9: The vertical positions of the fingertip in the frames

Step 3: *Find the pre-action frame.*

For a sequence of frames grouped for a click action, RoScript finds the frame with the fingertip vertical position equals to 0 and k-frames closely before a frame with the fingertip vertical position greater than 0 as the pre-action frame. The pre-action frame holds a snapshot of the touch screen before performing the click.

Step 4: *Locate the click position.*

In the fourth step, for each group of frames corresponding to a click action, RoScript finds the frame with the maximum value in the fingertip vertical position. This frame records a snapshot for the time point when the finger touches the screen. Let the fingertip's position be $\langle x_t, y_t\rangle$ in the found frame. For the click action, we calculate the exact click position from the fingertip position by:

$$P_{click} = \langle x_c, y_c\rangle = \langle x_t, y_t - \delta\rangle.$$

δ is a constant modelling the distance from the top of the fingertip (fingertip position) to the center touching position.

Step 5: *Crop an image for the clicked widget.*

With the pre-action frame and the click position, RoScript uses Algorithm 1 to crop an image for the clicked widget and thus generate test code for the click action. For the example click in Fig. 8(a), the generated test code is *t.click*( ).

The algorithm firstly does median blur operation [41] on a pre-action frame to optimize it for widget contour detection. After that, a contour detection like that used in the screenshot cropping (Section 6.1) is applied (line 3-5). We process the rectangle regions of the detected contours. If a contour region is too small, it may miss some clickable area in the widget, e.g., detecting a red rectangle region  for a borderless back button  . We slightly enlarge the region for a length ε on each direction so that it will cover more about the widget area (line 9-11, where LB_{real} is a threshold size in the real world, and we turn such a size into a size on the image frame by using the scale rate ρ_{sr} between the screenshot space and the real world obtained following Section 6.2). The algorithm then collects the contour regions covering the clicked position and with sizes smaller than an upper threshold (line 12-14, where MAX_{real} is the upper bound size in the real world). The upper

bound is used because too large cropped images can be confusing for understanding the GUI actions and they may also affect the widget click accuracy. Among the collected regions, a maximum size region (not a region of an internal shape inside a widget) is heuristically regarded as the clicked widget region. We crop widget images according to this region to generate test code.

The existing tool Sikuli [16] can also automatically crop widget images for script generation. However, it crops a fixed-size image to represent a target widget. The cropped result is easy to miss important parts of the clicked widget or cover the contents of other widgets, causing the generated test code difficult to understand. Our contour detection based method suffers less from that problem. As validated by the experiments (Section 8.2), in most cases, it can crop meaningful widget images for test script creation.

When there are non-click actions in a human action video, e.g., swipes or drags, RoScript may falsely recognize such actions as clicks. In that case, we need to manually replace the falsely recorded code with correct code for non-click actions. RoScript provides semi-automatic script recording for this purpose. The recording firstly uses the technique in Step 1 of the automatic recording to get a sequence of non-duplicated image frames with no finger on them. RoScript asks a user to specify the action types and touch positions on such frames. It then uses the technique in Step 5 of the automatic recording to crop images for the touched targets. With the manually specified action types and the automatically cropped images, test code can be semi-automatically generated.

Algorithm 1: Crop the widget image for a click action

Input: F – the pre-action frame,
P – the clicked position
Output: W – the image of the clicked widget

```
begin
    F' := medianBlur(F) ;
    Edges := cannyEdgeDetection(F') ;
    Edges := closingOperation(Edges) ;
    Contours := getAllContours(Edges) ;
    Rects := ∅ ;
    foreach C in Contours do
        R := getRectangleRegionOf(C) ;
        if size(R) < LB_real × ρ_sr^2 then
            R := enlarge(R, ε) ;
        end
        if (size(R) < MAX_real × ρ_sr^2) ∧ (P inside R) then
            Rects := Rects ∪ {R} ;
        end
    end
    R := getMaxSizeRegion(Rects) ;
    W := getRegionImage(F, R) ;
end
```

8 EVALUATION

The flexibility, accuracy, and efficiency of the test automation and the accuracy of the script recording are four key attributes of a usable test automation system. We conducted experiments on touch screen applications installed on different devices to evaluate these attributes of RoScript. The experiments answer the following research questions.

RQ1: How is the flexibility of RoScript in enabling test automation?

RQ2: How accurate can RoScript perform touch screen actions?

RQ3: How is the efficiency of RoScript in performing touch screen actions?

RQ4: How accurate can RoScript record click actions from human action videos?

RQ5: How does RoScript perform compared with other intrusive test automation techniques?

8.1 Experimental Setup

We automated the testing of 21 applications on 6 touch screen devices (Table 2-3) to answer RQ1. The devices cover mobile phones, a tablet, and embedded devices, with different sizes, colors, and operating systems. The applications cover areas of shopping, maps, messaging, games, office, etc. If a large number of test scenarios on these applications can be successfully automated (with successful test automation observed one or more times), that indicates RoScript is with good flexibility. We also carried out a testability analysis on different GUI widgets and actions to assess RoScript's ability in automating touch screen application testing.

Table 2: The Tested Touch Screen Devices

#	Model	Category	OS	Color	Size (in)
A	Samsung S5	Mobile Phone	Android 6.0.6	Black	5.1
B	Samsung Note4	Mobile Phone	Android 6.0.1	White	5.5
C	iPhone 5s	Mobile Phone	iOS 12.3.1	White	4.0
D	Telcast X98 Air	Tablet	Windows 8.1	White	9.7
E	Raspberry Pi 3B+	Embedded Dev.	Debian Linux	Black	5.0
F	GoPro Hero4	Embedded Dev.	Unknown	Black	1.9

Table 3: The Tested Touch Screen Applications

App	Description	Device
Taobao	an online shopping application	A
Ele.me	application of an online food delivery platform	A
Youdao Dict	a dictionary application	A
UC Browser	a mobile web browser	A
Tencent Video	a video streaming application	A
NetEase Music	a music application	B
Baidu Maps	a mapping application	B
WeChat	a messaging, social media, and mobile payment app	B
Galaxy Store	app store for Samsung devices	B
AndroidSys	the Android built-in system functionalities	B
Amazon	the Amazon mobile app	C
App Store	the Apple app store	C
Clock	a build-in clock app	C
FruitMatching	a connect lines puzzle game that matches fruits	C
iOS Sys	The iOS built-in system functionalities	C
Explorer	Windows file explorer	D
Firefox	a desktop web browser	D
Win8 Settings	the Windows control panel	D
RaspbianSys	the Raspbian built-in system functionalities	E
LibreOffice	LibreOffice Writer on Raspbian	E
GoPro	the GoPro in-device management system	F

For RQ2, we collected the success rates of accurately executing whole test scripts and single GUI actions to evaluate the accuracy of the robotic test automation. A test execution is said to be accurate if all the involved GUI actions are correctly triggered on the expected widgets. Due to the limitations of the used computer vision algorithms, the robot moving accuracy, and the unstableness

in camera-taken DUT photos, some actions in test scripts may occasionally fail to be accurately executed, while some may never be accurately executed. We run each test script 3 times to get the accuracy data with the unstableness in test execution considered.

For RQ3, the efficiency of RoScript in performing touch screen actions is mainly determined by the time consumed by photo taking, computer vision algorithms, and robot motions. We collected such time, together with the time spent on triggering a single GUI action and executing a whole test script, to evaluate the efficiency of the robotic testing approach.

To answer RQ4, we manually performed touch screen actions to execute the test scripts created for answering RQ1. The actions were recorded into videos, and we used automatic script recording to turn such videos back into test scripts again. The pre-action frames extracted from the videos, the identified click positions, and the cropped images were checked to evaluate the accuracy of RoScript's automatic script recording.

For RQ5, we selected Sikuli [16], a state-of-the-art intrusive visual GUI testing tool, as a baseline to evaluate RoScript, considering the accessibility and replicability of the existing techniques and tools on robotic test automation and visual GUI testing [3, 12, 16, 26, 33]. The flexibility, accuracy, and efficiency of the test automation and the accuracy of widget image cropping in test script recording were compared on the same test scenarios and settings to get a better understanding of the performance of RoScript.

In RoScript, the test engine is implemented in Python 3.6. We implement computer vision algorithms based on the basic functionalities provided by OpenCV 3.4. The test robot is driven by EBB commands [2] via USB connection. Besides the test robot, the whole system runs on a PC with Windows 10 OS, Intel i7-6700HQ CPU, and 16GB memory. We did experiments under a set of common parameter settings for the computer vision algorithms on different subject applications [7], without fine-tuning them for each individual subject. The parameters k, δ, ε, LB_{real}, and MAX_{real} used in Section 7 are 3, 4mm, 1.5mm, 55mm^2, and 400mm^2, respectively.

8.2 Results and Discussion

8.2.1 RQ1: The flexibility of the test automation. Table 5 presents the main experimental results. In the table, column *#script* shows the numbers of test scripts written for each subject application. Column *#action instruction* lists the numbers of total and key types of GUI action instructions included in the test scripts of an application, where CL, LP, SW, DG, LD, DC, and KP denote click, long press, swipe, drag, long press drag, double click, and keyboard press, respectively. Column *test accuracy* lists the success rates of accurately executing test scripts and GUI actions.

As shown in Table 5, we totally created 105 test scripts composed of 672 GUI action instructions for the subject applications. One test script was never accurately executed because of unstable touches on the Raspberry Pi's low sensitivity screen. 104 test scripts were at least be accurately executed by RoScript for once. These test scripts automate the testing of various application usage scenarios, including the simple ones like setting options by controlling standard widgets on Android-like systems particularly designed for touch screen usages and the more complex ones like renaming files on Windows-like systems not well designed for touch screen usages and playing games on non-standard graphical widgets [7]. The accurate execution of them suggests RoScript can automate a broad range of common testing activities on touch screen applications.

We also conducted a testability analysis to check the GUI widget and action combinations that can be automatically triggered by RoScript. The results are shown in Table 4, whose rows list the common semantic actions on widgets. These semantic actions can be triggered by raw actions like click, swipe, and long press in one or more ways. The raw actions are not listed here, as they are commonly supported regardless of the widget types. If a semantic action can be triggered, that means the test execution of its corresponding GUI functionality can be automated. In Table 4, an empty cell indicates a widget-action combination is considered uncommonly used in software implementation. Symbols △, ×, ○, ⦸, and ⊗ indicate different level of supports provided by RoScript.

From Table 4, we can see that for the 112 commonly used widget-action combinations (non-empty cells), RoScript theoretically supports testing 106 of them (94.6%). 6 of them are theoretically unsupported. The experiments covered 41 of these combinations, of which 35 were always successfully triggered, and 5 were successfully triggered for at least once. Many semantic actions like focusing on a tooltip and showing a context menu on a frame title were not covered because they are rarely used in touch screen interactions, although sometimes being implemented. Only one widget-action combination was covered but never successfully triggered because of unstable double clicks on a low sensitivity touch screen. These results indicate that RoScript provides very broad supports for GUI action triggering.

Some typical GUI actions theoretically unsupported by RoScript include focusing on a button and scaling up/down an image. These actions usually require keyboards (`Tab` key) or multi-finger touches to trigger. Besides, Table 4 only focuses on touch screen relevant actions. The actions requiring hardware buttons, sensors, etc. to trigger are also unsupported by RoScript. Even so, as a truly non-intrusive testing tool, RoScript is already very helpful for the test automation of various touch screen applications.

Table 4: GUI action triggering ability analysis

	border	button	checkbox	combo	context menu	drop-down	frame title	image	label	list	list/tree item	list/tree header	menu	menu item	progress bar	radio button	screen	scrollbar	slider	spinner	tab	text	toolbar	tooltip	tree	window
focus			×	×	△			△	△			△		△			△	×	△			△	×	○		△	△	△
show tooltip			⦸	△	△		△		△				△	△			△	△			△		△		△			
show cbt menu					△			△	○		△	△	○				△		△	△		△		○	△		△	△
select			⦸	○	○				○	○		△	⦸		○			○	○			○	△	○				
dbclick select			△					△	⦸				⊗				△		△			△		△				△
long press select			○						○				○						△									
deselect			△	○					○						△							△		△				
expand					△		○						○															
collapse					△		△						△															
move (drag)	○	○						△	○			△	⦸							○	○			△	△			○
text enter					△								△									△		○				
text delete					△								△									△		○				
scroll					○	△			○		○				△				○		○	△					△	○
home/back																			○									
scale up/down									×																			×

empty uncommon action
△ theoretically supported
× theoretically unsupported
○ theoretically supported, always successfully triggered in the experiments
⦸ theoretically supported, occasionally fail to be triggered in the experiments
⊗ theoretically supported, never successfully triggered in the experiments

Regarding the ability of verifying test results, RoScript applies the same image matching techniques used by traditional visual GUI testing tools to check the successfulness of test executions. According to [16], such result verification is highly usable for test automation.

Table 5: The Experimental Results

App	#script	#action instruction								test accuracy			test efficiency						script recording					Sikuli			
													script level time(s)			action level time(s)			click%	click record accuracy%							
		total	CL	LP	SW	DG	LD	DC	KP	#AAS	script%	action%	cv	motion	total	cv	motion	total		PAF	pos	crop	total	#script	acc%	time(s)	record%
Taobao	5	27	20	0	4	0	0	0	3	5	100.0	100.0	0.30	46.4	51.8	0.06	6.63	7.26	76.9	100	95.0	78.9	80.0	5	100	14.6	89.5
Ele.me	5	28	22	0	5	0	0	0	1	4	86.7	94.9	0.38	45.2	56.8	0.06	5.66	6.30	77.8	100	100.0	85.7	85.7	5	100	25.7	61.9
Youdao Dict	5	32	24	0	7	0	0	0	1	5	100.0	100.0	0.31	42.2	48.0	0.06	5.55	6.18	77.4	100	91.7	100.0	95.8	5	100	19.8	63.6
UC Browser	5	34	20	1	10	0	1	0	2	4	93.3	94.1	0.36	56.6	63.0	0.07	6.29	6.88	58.8	100	95.0	89.5	90.0	3	100	22.1	57.9
Tencent Video	5	26	21	0	4	0	0	0	1	5	100.0	100.0	0.35	36.9	43.8	0.06	4.73	5.36	80.8	100	90.5	84.2	76.2	5	100	25.1	63.2
NetEase Music	5	31	22	0	8	0	0	0	1	5	100.0	100.0	0.37	55.1	60.3	0.06	6.71	7.35	65.8	100	96.0	100.0	96.0	5	100	21.9	83.3
Baidu Maps	5	37	27	1	5	1	0	1	2	4	93.3	96.5	0.49	70.3	76.5	0.06	6.63	7.21	72.9	100	85.7	96.7	85.7	4	100	22.4	73.3
WeChat	5	39	28	1	6	0	0	0	4	3	80.0	85.5	0.36	72.2	79.7	0.06	8.21	8.83	71.8	100	92.9	88.5	82.1	4	100	21.1	69.2
Galaxy Store	5	32	23	0	8	0	0	0	1	5	100.0	100.0	0.43	51.4	57.4	0.06	5.71	6.38	71.9	100	100.0	95.7	95.7	5	100	27.4	69.6
AndroidSys	5	30	17	2	3	0	1	0	2	5	100.0	100.0	0.37	34.2	39.8	0.07	5.02	5.71	60.7	100	88.2	93.3	82.4	3	100	14.7	46.7
Amazon	5	51	34	0	13	0	0	0	4	2	73.3	86.0	0.52	86.8	109.7	0.03	4.26	4.80	68.0	100	97.1	90.9	88.2	5	60.0	63.6	42.4
App Store	5	31	23	0	7	0	0	0	1	5	100.0	100.0	0.28	48.3	55.2	0.04	4.93	5.43	74.2	100	95.7	90.9	87.0	5	100	21.9	45.5
Clock	5	33	25	0	2	2	0	0	1	4	93.3	96.0	0.34	52.9	58.2	0.04	5.63	6.19	77.1	100	100.0	100.0	100.0	5	80.0	21.0	70.4
FruitMatching	5	25	25	0	0	0	0	0	0	4	93.3	97.8	0.42	51.2	57.1	0.04	4.49	5.01	100	100	87.9	62.1	60.6	5	100	30.7	62.1
iOS Sys	5	27	18	1	3	1	0	0	3	5	100.0	100.0	0.18	41.7	47.2	0.04	5.79	6.42	72.0	100	94.4	94.1	88.9	4	100	28.6	58.8
Explorer	5	37	25	8	1	1	0	0	2	3	86.7	91.9	1.10	52.0	58.8	0.16	6.35	7.05	67.6	100	96.0	91.7	92.0	5	100	12.2	45.8
Firefox	5	36	27	5	0	1	0	0	3	5	100.0	100.0	1.07	70.5	79.5	0.14	8.60	9.33	73.0	100	70.4	94.7	77.8	5	100	24.4	57.9
Win8 Settings	5	31	23	2	2	0	0	0	0	5	100.0	100.0	0.86	41.4	46.5	0.15	5.76	6.44	74.2	100	87.0	95.0	87.0	4	100	16.3	60.0
RaspbianSys	5	28	22	0	0	2	0	1	0	1	60.0	75.0	0.52	32.6	37.4	0.09	4.94	5.64	84.6	100	90.9	75.0	68.2	5	80.0	7.9	60.0
LibreOffice	5	28	25	0	0	1	1	0	1	2	73.3	83.3	0.51	36.4	42.2	0.08	5.52	6.23	89.3	100	92.0	78.3	72.0	5	100	9.0	47.8
GoPro	5	29	20	0	9	0	0	0	0	4	86.7	85.6	0.12	41.5	45.9	0.03	6.11	6.75	66.7	100	100.0	90.0	90.0	0			50.0
Average											91.4	94.6	0.46	50.8	57.8	0.07	5.88	6.51	74.4	100	92.7	89.3	84.8		96.0	22.5	60.9
Total	105	672	491	21	97	9	3	2	33	85														92			

8.2.2 RQ2: The accuracy of the test automation. For the accuracy of performing touch screen actions, from columns *test accuracy*: *script%-action%* of Table 5, we can see that RoScript's average application-level success rates of accurately executing test scripts and accurately triggering GUI actions are 91.4% and 94.6%, respectively. The corresponding total success rates for all the test scripts and GUI actions are 91.4% and 94.4%. (In a test script, the actions after the first failure in accurately triggering a GUI action were counted as inaccurately triggered.) Column *test accuracy*: *#AAS* of Table 5 shows that in total, 85 (81.0%) test scripts were always accurately executed. These data suggest RoScript can automate the test activities on a touch screen application with high accuracy.

RoScript got the worst accuracy on Raspberry Pi, which has a low sensitivity touch screen and runs a Debian Linux based system not well tuned for touch screen usages. The common reasons causing a GUI action failing to be accurately triggered include: (1) too light or heavy touches on screens with different touch sensitivities; (2) the low template matching accuracy of some too small and unclear widget images and some widget images containing light-colored texts; and (3) the Moiré pattern noise on some screen photos. For the touch sensitivity relevant failures, better touch mechanisms and more compatible stylus pens may be needed. For the other failures, a possible way to improve the test automation accuracy is to fine-tune the camera shooting environments and configurations for a subject to obtain better camera photos of a touch screen. The experiments tested different subjects under the same camera shooting environments and configurations and had not tried such tuning. If doing fine-tuning individually for each subject, even better test automation accuracy might be achieved.

8.2.3 RQ3: The efficiency of the test automation . RoScript takes about 0.3-0.7s to obtain a photo of the DUT. Column *test efficiency* of Table 5 lists some other test-script-level and GUI-action-level efficiency data about our robotic testing. Sub-columns *cv* and *motion* show the time consumed by computer vision algorithms and robot motions, respectively, when executing a test script or a GUI action. Sub-column *total* lists the total time spent on executing a test script or triggering a GUI action.

The efficiency data in Table 5 show that under RoScript, each GUI action averagely takes 0.07s in running computer vision algorithms. That time is almost ignorable. Triggering a GUI action via the robot averagely costs 6.5s. At the test script level, running a test script consisting of on average 6.4 GUI action instructions and some other code including waits and sleeps averagely costs 57.8s. Such a test execution speed is acceptable in practice.

8.2.4 RQ4: The accuracy of the test script recording. Column *script recording* in Table 5 presents some results about the automatic script recording. Its sub-column *click%* lists the percentage of click actions in the human action videos of each application [7]. Sub-column *PAF* shows the accuracy (the success rate of accurately doing something) of the pre-action frame extraction for the clicks. Sub-column *pos* lists the accuracy of click position identification. A click position is considered accurately identified if clicking there can correctly trigger the expected GUI action. Sub-column *crop* shows the accuracy of widget image cropping for clicks with positions accurately identified. We consider a widget image is accurately cropped if the cropped image covers the distinguishing part of the clicked widget, is easy for understanding (the image shall not contain parts of other unclicked widgets, and if the clicked target is a string, then at least a whole word shall be included), and is with target action triggerable on the image center. Sub-column *total* lists the overall accuracy of click action recording. We consider a click action is accurately recorded if finally its widget image is accurately got.

From Table 5, we can see that for the click actions, at the application level, their average accuracy of pre-action frame extraction, click position identification, widget image cropping, and overall click action recording are 100%, 92.7%, 89.3%, and 84.8%, respectively. The overall click action recording accuracy is about 2% higher than the multiplication of the click position identification accuracy and the widget image cropping accuracy (on average 82.8%). This is because sometimes the recognized widget contour regions were a little larger than the real widgets, which can tolerate slight errors in the detected click positions. The total recording accuracy for all clicks corresponding to the 491 click instructions is 84.5%. That means the click actions can be recorded in very high accuracy. Click

actions occur very frequently on touch screen applications (e.g., taking about 74.4% in all the involved GUI actions in the studied test scenarios). With that accuracy, RoScript's test script creation can be largely automated with the script recording technique.

In the recording, a click position fails to be accurately identified often because of the skin-like color images on a touch screen which cause the hand contours difficult to be accurately recognized and the perspective effect in camera shots which causes the touch positions hard to be accurately located. The common situations that a widget image fails to be accurately cropped include: a clicked widget is too small, and its neighbor widgets also occur in the cropped result; and a clicked widget is too big, and the cropped image does not cover all its content. The script recording got the worst accuracy on the *FruitMatching* application which contains many closely-spaced small fruit image widgets challenging for click position identification and image cropping. Better script recording techniques are still under design to further increase the recording accuracy.

8.2.5 RQ5: The comparison with Sikuli. For the flexibility of test automation, on platforms like Windows and Linux natively supported by Sikuli, RoScript's GUI action triggering ability is weaker than that of Sikuli since Sikuli can trigger almost all possible actions via OS facilities. RoScript's test result verification ability is also a little weaker, because unlike Sikuli, it relies on inaccurate camera photos, instead of accurately captured screenshots, for result checking. On platforms like Android/iOS testable via screen mirroring but not natively supported by Sikuli, RoScript cannot guarantee accurate test automation for a few test scenarios. Sikuli is not directly usable when there are some mobile gestures originally unsupported by it (see the script numbers less than that of RoScript from column *Sikuli: #script* of Table 5). Under such circumstances, the flexibility of these two tools is somewhat comparable. On arbitrary touch screen platforms like GoPro, RoScript certainly has better flexibility because Sikuli's intrusive testing cannot work on such platforms.

For the accuracy of test automation, at the test script level, Sikuli achieved a test automation accuracy of 96% (see column *Sikuli: acc%* of Table 5. There were also a few inaccurate test script executions due to Sikuli's own technique limitations). RoScript's test automation accuracy (91.4%) is lower than that of Sikuli because its camera-taken GUI states and robot movements are not accurate. Even so, considering that the test automation of RoScript is completely non-intrusive and applicable to almost all possible touch screen applications, such sacrifice in test automation accuracy might be worthy.

For the efficiency of test automation, Sikuli constantly took about 0.5s to trigger a GUI action. This is much faster than RoScript (6.5s on average). However, taking the time spent on waiting for GUI states into account, at the test script level, RoScript and Sikuli's average test execution time still remains in the same order of magnitude (RoScript: 57.8s, Sikuli: 22.5s, see column *Sikuli: time* of Table 5).

For script recording, Sikuli cannot non-intrusively record test scripts. On the same settings as that of RoScript, the average accuracy of Sikuli's fixed-size image cropping (with the desktop icon size as the fixed size) is about 60.9% (see column *Sikuli: record%* of Table 5). RoScript's widget image cropping accuracy (89.3%) is much higher than that of Sikuli. Besides, RoScript cropped more beautiful widget images, since its cropped images usually are compact and show widgets in the center, while Sikuli's image cropping results are mostly not compact and often show widgets at the corners.

8.3 Threats to Validity

There are three major validity threats in the experiments. (1) The first is that the use of the devices, applications, and test scenarios in the experiment is limited, which might limit the generalization of the experimental results. Nevertheless, we have evaluated RoScript on 6 representative devices, 21 applications from different areas, and 105 test scenarios with 672 GUI actions. With such a number of subjects, we believe it is convincing to draw some effective conclusions. (2) The second threat is the experimental environment, such as the camera shooting environment and configurations. We conducted the experiments in a normal laboratory room, with a camera resolution of 1600x1200 and a common camera shooting distance enough to have all the tested devices fully visible. Such an environment and configurations were not fine-tuned for different devices. In practical testing, by fine-tuning such factors, even better test automation accuracy or efficiency might be achieved. (3) The third threat is that the quality of the recorded human action videos may have impact on the script recording accuracy. To alleviate the impact, we proposed a set of video recording instructions (Section 7) to guide the experiments. All experiments were done strictly following these instructions. We have already tested the recording on 105 videos covering 491 click instructions, and the experience shows that by following the video recording instructions, high script recording accuracy can be achieved.

9 CONCLUSION AND FUTURE WORK

This paper presents a visual test script driven robotic testing system to automate the testing of touch screen applications. It uses physical robots to conduct fully non-intrusive testing and leverages computer vision algorithms to drive test execution. A video-based approach is also proposed to automatically record human touch screen click actions into test scripts, which can reduce the test script creation cost. The experiments on various touch screen devices and applications show that the approach is usable.

There are still some limitations in the current implementation of RoScript. First, due to the limitations of physical robots, its test execution speed is slower than traditional intrusive testing. Second, some unstable screen touches and low-quality camera pictures may affect the accuracy of performing GUI actions. Third, the approach yet does not support automatically recording arbitrary GUI actions. Fourth, RoScript only supports testing devices with sizes smaller than A4 paper. These may affect the scalability, robustness, and application scope of the approach. In the future, we plan to continuously improve the testing system to address these limitations.

ACKNOWLEDGMENTS

This work is partially supported by the National Key R&D Program of China (2018YFB1003900), the National Natural Science Foundation of China (61872177), and the Equipment Pre-Research Project of China (41402020501-170441402030). Ju Qian and Lin Chen are the corresponding authors.

REFERENCES

[1] 2020. *Appium: Automation for iOS and Android apps.* http://appium.io
[2] 2020. *EBB Command Set.* http://evil-mad.github.io/EggBot/ebb.html
[3] 2020. *EggPlant.* https://www.eggplantsoftware.com/
[4] 2020. *OpenCV.* https://opencv.org/
[5] 2020. *OptoFidelity test solutions.* https://www.optofidelity.com/test-solutions/
[6] 2020. *Robotium.* https://github.com/RobotiumTech/robotium
[7] 2020. *RoScript evaluation artifacts.* https://github.com/juqian/ICSE2020-RoScript
[8] 2020. *UIAutomator.* http://developer.android.com/tools/testingsupport-library
[9] 2020. *VNC.* https://en.wikipedia.org/wiki/Virtual_Network_Computing
[10] Emil Alégroth and Robert Feldt. 2017. On the long-term use of visual GUI testing in industrial practice: a case study. *Empirical Software Engineering* 22, 6 (2017), 2937–2971.
[11] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. 2015. Visual GUI testing in practice: challenges, problems and limitations. *Empirical Software Engineering* 20, 3 (2015), 694–744.
[12] Emil Alégroth, Michel Nass, and Helena H. Olsson. 2013. JAutomate: A tool for system and acceptance test automation. In *IEEE 6th International Conference on Software Testing, Verification and Validation (ICST)*. 439–446.
[13] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2014), 53–59.
[14] Andreas Bihlmaier and Heinz Wörn. 2014. Robot unit testing. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 255–266.
[15] Emil Borjesson and Robert Feldt. 2012. Automated system testing using visual GUI testing tools: A comparative study in industry. In *IEEE 5th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 350–359.
[16] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. 2010. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1535–1544.
[17] Mathieu Collet, Arnaud Gotlieb, Nadjib Lazaar, and Morten Mossige. 2019. Stress testing of single-arm robots through constraint-based generation of continuous trajectories. In *IEEE International Conference On Artificial Intelligence Testing*. IEEE, 121–128.
[18] Karthikeyan B. Dhanapal, K Sai Deepak, Saurabh Sharma, Sagar P. Joglekar, Aditya Narang, Aditya Vashistha, Paras Salunkhe, Harikrishna G.N. Rai, Arun A. Somasundara, and Sanjoy Paul. 2012. An innovative system for remote and automated testing of mobile phone applications. In *Annual SRII Global Conference*. IEEE, 44–54.
[19] Mattia Fazzini, Eduardo Noronha de A Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A technique for recording, encoding, and running platform independent Android tests. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 149–160.
[20] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing-and touch-sensitive record and replay for Android. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 72–81.
[21] 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 *the 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.
[22] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. SARA: self-replay augmented record and replay for Android in industrial cases. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 90–100.
[23] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented Android ecosystem. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 215–224.
[24] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 349–366.
[25] Hengle Jiang, Sebastian Elbaum, and Carrick Detweiler. 2017. Inferring and monitoring invariants in robotic systems. *Autonomous Robots* 41, 4 (2017), 1027–1046.
[26] Teemu Kanstrén, Pekka Aho, Arttu Lämsä, Henar Martin, Jussi Liikka, and Miska Seppänen. 2015. Robot-assisted smartphone performance testing. In *IEEE International Conference on Technologies for Practical Robot Applications*. IEEE, 1–6.
[27] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.
[28] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for Android: are we there yet in industrial cases?. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 854–859.
[29] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 399–410.
[30] Jin-Feng Luan, Dror Saaroni, and Xiao-Ming Hu. 2018. Generating software test script from video. US Patent 10,019,346.
[31] Niloofar Mansoor, Jonathan A Saddler, Bruno Silva, Hamid Bagheri, Myra B Cohen, and Shane Farritor. 2018. Modeling and testing a family of surgical robots: an experience report. In *the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. ACM, 785–790.
[32] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 16–26.
[33] Ke Mao, Mark Harman, and Yue Jia. 2017. Robotic testing of mobile apps for truly black-box automation. *IEEE Software* 34, 2 (2017), 11–16.
[34] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. 2019. Specification patterns for robotic missions. *arXiv preprint arXiv:1901.02077* (2019).
[35] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of Android applications. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 559–570.
[36] Clemens Mühlbacher, Gerald Steinbauer, Michael Reip, and Stephan Gspandl. 2019. Constraint-based testing of an industrial multi-robot navigation system. In *IEEE International Conference On Artificial Intelligence Testing*. IEEE, 129–137.
[37] Sinan Naji, Hamid A. Jalab, and Sameem A. Kareem. 2019. A survey on skin detection in colored images. *Artificial Intelligence Review* 52, 2 (2019), 1041–1087.
[38] Shaul Oron, Tali Dekel, Tianfan Xue, William T Freeman, and Shai Avidan. 2017. Best-buddies similarity—Robust template matching using mutual nearest neighbors. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 8 (2017), 1799–1813.
[39] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: A remote execution based record-and-replay tool for mobile applications. In *the 38th International Conference on Software Engineering (ICSE)*. ACM, 571–582.
[40] Paridhi Swaroop and Neelam Sharma. 2016. An overview of various template matching methodologies in image processing. *International Journal of Computer Applications* 153, 10 (2016), 8–14.
[41] Richard Szeliski. 2010. *Computer vision: algorithms and applications* (1st ed.). Springer-Verlag, Berlin, Heidelberg.
[42] Ragnar Wernersson. 2015. *Robot control and computer vision for automated test system on touch display products.* Master's thesis. Lund University, Sweden.
[43] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 307–317.
[44] Rahulkrishna Yandrapally, Giriprasad Sridhara, and Saurabh Sinha. 2015. Automated modularization of GUI test cases. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE, 44–54.