

GUI Testing Using Computer Vision

Tsung-Hsiang Chang
MIT CSAIL
vgod@mit.edu

Tom Yeh
UMIACS & HCIL
University of Maryland
tomyeh@umiacs.umd.edu

Robert C. Miller
MIT CSAIL
rcm@mit.edu

ABSTRACT

Testing a GUI's visual behavior typically requires human testers to interact with the GUI and to observe whether the expected results of interaction are presented. This paper presents a new approach to GUI testing using computer vision for testers to automate their tasks. Testers can write a visual test script that uses images to specify which GUI components to interact with and what visual feedback to be observed. Testers can also generate visual test scripts by demonstration. By recording both input events and screen images, it is possible to extract the images of components interacted with and the visual feedback seen by the demonstrator, and generate a visual test script automatically. We show that a variety of GUI behavior can be tested using this approach. Also, we show how this approach can facilitate good testing practices such as unit testing, regression testing, and test-driven development.

Author Keywords

GUI testing, GUI automation, test by demonstration

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces—*Graphical user interfaces (GUI)*; D.2.5 Software Engineering: Testing and Debugging—*Testing tools*

General Terms

Algorithms, Design, Reliability

INTRODUCTION

Quality Assurance (QA) testers are critical to the development of a GUI application. Working closely with both programmers and designers, QA testers make efforts to ensure the GUI application is correctly implemented by the former following the design specification drawn by the latter. Without such efforts, there is no guarantee the usability promised by a good design is fully realized in the implementation.

However, GUI testing is a labor intensive task. Consider the following GUI behavior defined in a design specification of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHI 2010, April 10 – 15, 2010, Atlanta, Georgia, USA

Copyright 2010 ACM 978-1-60558-929-9/10/04...\$10.00.



Figure 1. GUI testing (left) traditionally requires human testers to operate the GUI and verify its behavior visually. Our new testing framework allows the testers to write visual scripts (right) to automate this labor-intensive task.

a video player: *click the button*  *and it becomes* . To test if this behavior is correctly implemented, a tester must *look for* the “play” button on the screen, *click* on it, and *see* if it is replaced by the “pause” button. Every time this behavior needs to be tested again, the tester must manually repeat the same task all over again.

While GUI testers often toil in their tedious tasks, testers of non-GUI applications have been enjoying the convenience of tools to automate their tasks. For example, to test if the function call `addOne(3)` behaves correctly, a tester can write a script that makes this function call, followed by an assertion function call, such as `assert(addOne(3) == 4)`, to check if the result is equal to 4 and report an error if not. This script can be run automatically as many times as desired, which greatly reduces the tester's effort.

In this paper, we present Sikuli Test, a new approach to GUI testing that uses computer vision to help GUI testers automate their tasks. Sikuli Test enables GUI testers to write *visual* scripts using images to define what GUI widgets to be tested and what visual feedback to be observed. For example, to automate the task of testing the behavior of the video player described above, a tester can write the following script:

```
click(); assertExist(); assertNotExist();
```

When this script is executed, it will act like a robotic tester with eyes to *look for* the “play” button on the screen, *click* on it, and *see* if it is replaced by the “pause” button, as if the human tester is operating and observing the GUI him- or herself (Figure 1).

We make the following contributions in this paper:

Interview study with GUI testers We examine the limitations of current testing tools and suggest design requirements for a new testing framework.

Automation of visual assertion Based on the visual automation API provided by Sikuli Script [18], a set of visual assertion API is added to determine if expected outputs are shown or not. The extension of visual assertion fulfills the automation of GUI testing by using images for verifying outputs in addition to directing inputs.

Test-By-Demonstration Testers can interact with a GUI and record the actions they perform and visual feedback they see. Test scripts can be automatically generated to reproduce the actions and verify the visual feedback for testing purposes.

Support of good testing practices Features are introduced to support good testing practices including unit testing, regression testing, and test driven development.

Comprehensive evaluation We analyze the testability of a wide range of visual behavior based on five actual GUI applications. Also, we examine the reusability of test scripts based on two actual GUI applications evolving over many versions.

INTERVIEW STUDY

To guide the design and development of our new GUI testing tool, we conducted informal interviews with four professionals of GUI testing from academia and industry. Questions asked during the interviews were centered on three topics: current testing practices, use of existing tools, and experience with existing tools.

In terms of testing practices, we found most of our subjects are involved in the early design process to coordinate and formulate workable test plans to ensure quality and testability. Testing is performed frequently (often daily) on the core components. For example, underlying APIs are tested with simulated inputs and checked if they produce expected outputs. But testing the outward behavior of GUIs is less frequent, usually on major milestones by a lot of human testers. Some of them regularly apply good testing practices such as unit testing, regression testing, and test-driven development; but the scope of these practices is limited to the parts without GUI.

In terms of the use of testing tools, some have developed customized automation tools. They write scripts that refer to GUI objects by pre-programmed names or by locations to simulate user interactions with these objects. Some have been using existing tools such as Autoit [1], a BASIC-like scripting language designed to automate user interactions for Windows GUI applications.

In terms of experience with these tools, our subjects expressed frustration and described their experience as sometimes “painful”, “slow”, and “too much manual work.” Several problems with current automatic testing tools were iden-

tified by the subjects, which might explain this frustration. First, whenever the GUI design is modified and the positions of GUI components are rearranged, automatic tools based on the absolute position of components often fail and would actually “slow down the testing process” because of the need to modify the test scripts. Second, while automatic tools based on component naming may avoid this problem, many components simply can not or have not been named.

Based on the findings of this interview, we identified the following five design goals to guide the design and development of our new GUI testing tool:

- (G1) The tool should allow testers to write scripts to automate tests.
- (G2) The tool should not require testers to refer GUI components by names or by locations.
- (G3) The tool should minimize the instances when test scripts need to be modified due to design changes.
- (G4) The tool should minimize the effort of writing test scripts.
- (G5) The tool should support good testing practices such as unit testing, regression testing, and test-driven development.

TESTING BY VISUAL AUTOMATION

We present Sikuli Test, a testing framework based on computer vision that enables developers and QA testers to automate GUI testing tasks. Consider the following task description for testing a particular GUI feature:

Click on the color palette button. Check if the color picking dialog appears.

To carry out this test case, QA testers need to manually interact with the GUI and visually check if the outcome is correct. Using Sikuli Test, the testers can automate this process by converting the task description into an automation script. This script consists of action statements to simulate the interactions and assertion statements to visually verify the outcomes of these interactions. For example, the above task description can be easily translated into a test script as:

```
click(); assertExist(
```

By taking this image-based scripting approach, Sikuli Test meets the first three design goals: it allows testers to write visual scripts to automate tests (G1), to refer to GUI objects by their visual representation directly (G2), and to provide robustness to changes in spatial arrangements of GUI components (G3). The details of how to write test scripts using action statements and assertion statements are given next.

Simulating Interactions using Action Statements

To simulate interactions involved in a test case, QA testers can write action statements using the API defined in Sikuli

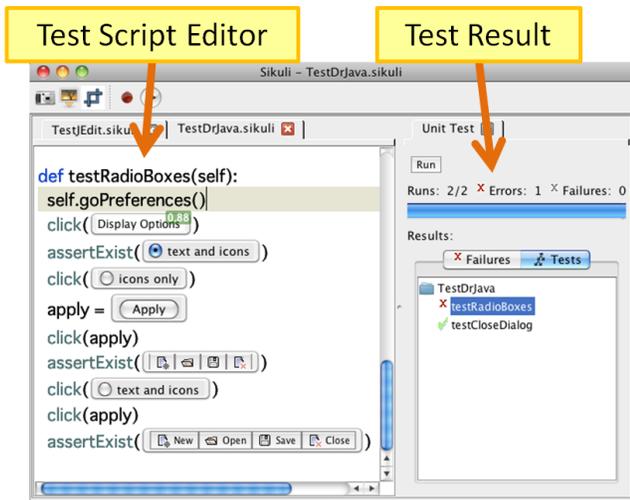


Figure 2. Sikuli Test interface consists of a test script editor and an information panel summarizing the test result.

Script [18]. Sikuli Script is a visual automation system that provides a library of functions to automate user inputs such as mouse clicks and keystrokes. These functions take screenshots of GUI components as arguments. Given the image of a component, Sikuli Script searches the whole screen for the component to deliver the actions. For example,

- `click()` clicks on the close button,
- `dragDrop(, )` drags a word document to the trash,
- `type(, "CHI")` types "CHI" in a search box.

Since Sikuli Script is based on a full scripting language, Python, it is possible for QA testers to programmatically simulate a large variety of user interactions, simple or complex.

Verifying Outcomes using Visual Assertion Statements

Sikuli Test introduces two *visual assertion* functions. QA testers can include these functions in a test script to verify whether certain GUI interaction generates the desired visual feedback. These two assertion functions are:

`assertExist(image or string [, region])`
asserts that an image or string that should appear on screen or in a specific screen region

`assertNotExist(image or string [, region])`
asserts that an image or a string should not appear on screen or in a specific screen region

The image is specified as URL or a path to an image file. It also can be captured by a screenshot tool provided in our Integrated Development Environment (IDE). When a string

is specified, OCR (Optical Character Recognition) is performed to check if the specified string can be found in the screen region. The optional parameter `region` is specified as a rectangular area on the screen (i.e., x, y, width, height). If not specified, the entire screen is checked. Alternatively, the region can be specified as a second image, in which case the entire screen is searched for that image and the matching region is searched for the first image. Spatial operators such as *inside*, *outside*, *right*, *bottom*, *left*, and *top* can be further applied to a region object to derive other regions in a relative manner.

Examples

We present examples to illustrate how test scripts can be written to verify visual feedback.

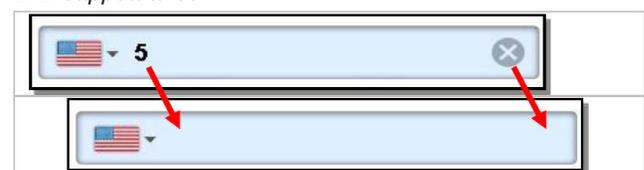
1. Appearance



```
1 type(":p")
2 assertExist()
```

In some instant messengers, textual emoticons, e.g. smiley face :), are replaced by graphical representations automatically. This example shows how to test the appearance of the corresponding graphical face once the textual emoticon is entered in Windows Live Messenger.

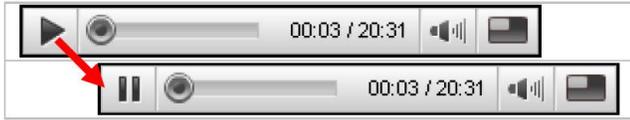
2. Disappearance



```
1 blueArea = find() [0]
2 closeButton = 
3 click(closeButton)
4 assertNotExist(closeButton, blueArea)
5 assertNotExist("5", blueArea)
```

In this example, the close button  is expected to clear the content of the text box as well as itself. Suppose the GUI is already in a state that contains a "5", at first we find the blue text box on the screen and store the matched region that has the highest similarity in `blueArea`. Then, after clicking the close button, two `assertNotExist` are used to verify the disappearance in the blue area.

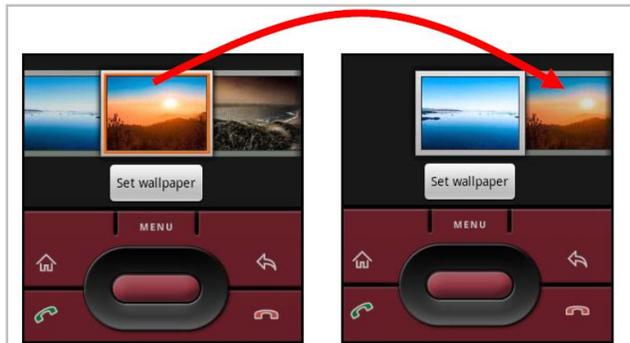
3. Replacement



```
1 click (▶)
2 assertNotExist (▶)
3 click (⏸)
4 assertNotExist (⏸)
5 assertExist (▶)
```

Typical media players have a toggle button that displays the two possible states of the player, playing or pause. In this example, we demonstrate a test case that tests the typical toggle button on youtube.com, a popular website of video collection. This script clicks on the play button first, and asserts the disappearance of it. A trick here is that we do not need an `assertExist` for the pause button, because the second click implicitly asserts the existence of the pause button. If the play button is not replaced by the pause button as expected, an error will be thrown from the click statement.

4. Scrolling/Movement



```
1 sunset = 
2 old_x = find(sunset)[0].x
3 click (⏪)
4 assert (find(sunset)[0].x > old_x)
```

Since Sikuli Test is independent of any GUI platform, it also can be used to test mobile applications running on an emulator. This example shows how to test scrolling and movement on an Android emulator. This test case works by comparing the position of the target before and after an action that should move the target. After clicking on the left button, we expect the series of images to scroll rightward. Therefore, the new x coordinate should be larger than the old one. We choose the image of sunset to be the target. Its x coordinate that derived from the most similar match of `find()` is stored in `old_x`. After the clicking on the left button, its new x coordinate derived from `find()` again is used to be compared with the `old_x` for verifying the correctness of the implementation.

TESTING BY DEMONSTRATION

Sikuli Test provides a record-playback utility that enables QA testers to automate GUI testing by demonstration. The operation of a GUI can be described as a cycle consisting of actions and feedback. Given a test case, the testers follow the given actions to operate a GUI, and verify if the visual feedback is the same as expected. If so, they proceed to do the next actions and to verify further feedback.

With the record-playback mechanism, the testers can demonstrate the interactions involved in the test case. The actions as well as the screen are recorded and translated into a sequence of action and assertion statements automatically. The action statements, when being executed, can replicate the actions, as if the testers are operating the GUI themselves. Besides, the assertion statements can verify if the automated interactions lead to the desired visual feedback, as if the testers are looking at the screen themselves.

The test-by-demonstration capability of Sikuli Script satisfies the design goal of minimizing the effort needed to write test scripts (G4). Details of how demonstration is recorded and how actions and assertions are automatically generated from the recorded demonstration will be given next.

Recording Demonstration

As QA testers demonstrate a test case, a recorder is running in the background to capture the actions they perform and the visual feedback they see. To capture actions, the recorder hooks into the global event queue of the operating system to listen for input events related to the mouse and the keyboard. The list of mouse events recorded includes `mouse_down`, `mouse_up`, `mouse_move`, and `mouse_drag`. Each mouse event is stored with the cursor location (x, y) and the state of buttons. The keyboard events recorded are includes `key_down` and `key_up`, stored together with key codes. All events include a `timestamp` that is used to synchronize with the screen recording. To capture screens, the recorder grabs the screenshot of the entire screen from the video buffer in the operating system periodically. In our prototype, the recording can be done at 5 fps at a resolution of 1280x800 on a machine with 2Ghz CPU and 2GB memory.

Generating Action Statements

Given a series of screen images and input events captured by the recorder, action statements can be generated to replay the interactions demonstrated by the testers. For example, a single mouse click recorded at time t at location (x, y) can be directly mapped to `click(I)` where I is the image of the GUI component that was clicked. The image I can be obtained by cropping a region around (x, y) from the screen image captured at time $t - 1$ right before the click event. In our current implementation, a constant-size (80x50) region around the input location is cropped to represent the target GUI component receiving the input. Even though the region may not necessarily fit the target component perfectly, often it contains enough pixels to uniquely identify the component on the screen. If ambiguity arises, the user can adjust the cropping area to include more pixels of the component or the context to resolve the ambiguity at any time.

To execute an action statement, the automation engine visually identifies the target GUI component's current location (x', y') by searching the current screen for an image region matching the target image I . To find a given pattern, we apply the template matching technique with the normalized correlation coefficient implemented in OpenCV in our current system [18]. This technique treats the pattern as a template and compares the template to each region with the same size in an input image to find the region most similar to the template. Then, the click event is delivered to the center of the matched region to simulate the desired user interaction.

Some input events may need to be grouped into a single action statement. For example, two consecutive mouse clicks in a short span of time is mapped to `doubleClick()`. Keyboard typing events can be clustered to form a string and mapped to `type(string)`. A `mouse_down` event at one location followed by a `mouse_up` event at another location can be mapped to `dragDrop(I,J)` where I and J denote the images extracted from the locations of the two mouse events respectively.

Generating Assertion Statements

Assertion statements can also be automatically derived from the screen images captured during the demonstration. We developed and implemented a simple vision algorithm to accomplish this. We assume any salient change between the two images is very likely to be the visual feedback caused by an input event. Our algorithm compares the screen images I_t and I_{t+1} where t is the time of a recorded input event, and identifies pixels that are visually different. It then clusters the changed pixels in close proximity and merges them into the same group. Each group of pixels would probably correspond to the same GUI component. Finally, it computes a bounding rectangle around each group and obtains a cropped image containing the visual feedback of each GUI component visually affected by the input event.

An assertion statement that can be later used to check the presence of the visual feedback can be generated with this algorithm. Figure 3 shows an example of deriving the visual feedback where a drop-down box is opened by clicking. Often, more than one GUI component can exhibit visual feedback as the result of a single input event. In this case, our algorithm results in a compound assertion statement including multiple cropped image regions. For example, Figure 4 shows a dialog box with a checkbox that can be used to enable several GUI components at once. Checking this checkbox will cause all previously greyed out components in a panel to regain their vivid colors.

An optional step for the tester to increase the reliability of the automatic visual feedback detector is to provide hints to where it should look for the visual feedback. After performing an interaction and before moving on to the next, the tester can move the mouse cursor to the area where the visual feedback has occurred and press a special key, say F5, to trigger a hint. The detector can use the location of the cursor to extract the relevant visual feedback more reliably and generates an appropriate assertion statement.



Figure 4. Example of automatic generation of assertion statements from detected visual feedback.

While we can identify many cases in which visual assertion statements can be created automatically in this manner, there remain a few challenges. First, periodic changes in the desktop background, such as those related to the system clock or the wireless signal indicator, may be inadvertently detected but irrelevant to the GUI to be tested. One solution would be to ask the testers to specify the boundary of the GUI beforehand so that background noises can be filtered out. Second, certain actions might take longer to obtain any visual feedback; the screen image captured immediately after the action might not contain the visual feedback. One solution would be to wait until a significant change is detected. Third, some visual feedback may involve animation spanning several frames, for example, a large window appearing in a blind-rolling-down fashion. One solution would be to wait until the screen has stabilized and focus only on the final visual feedback. However, while it is possible to test the final feedback, testing the intermediate steps of an animation can still be unreliable, because it is difficult to synchronize between the frames sampled during the demonstration time and those sampled during the test time.

SUPPORTING GOOD TESTING PRACTICES

Sikuli Test comes with a set of features to help GUI developers and QA testers engage in good testing practices such as unit testing, regression testing, and test-driven development, satisfying the last design goal (G5).

Unit Testing

When a GUI is complex, to make sure it is tested thoroughly requires a systematic approach. One such approach is to break the GUI down into manageable units, each of which targets a particular part, feature, or scenario. This approach is known as *unit testing*.

To support unit testing for GUI, Sikuli Test draws many design inspirations from JUnit, a popular unit testing framework for Java programming:

1. Testers can define each test as a function written in Python. Every test function is meant to be run independently with-



Figure 3. An example of taking the difference between two screens to derive the visual feedback automatically

out relying on the side-effects of another test function. For example, after testing the *exit* button, which has the side effect of closing the application, no more tests can be run unless the GUI is restarted. Therefore, to run every test independently, Sikuli Test provides two functions *setUp()* and *tearDown()* that can be overridden by testers to set up and to clean up the testing environment. A typical way to achieve the independence is always starting the GUI in a fresh configuration before running a test.

2. Testers can define common action functions to automatically advance the GUI to a particular state in order to run certain tests only relevant in that state. Common action functions can be shared among all test cases in the same script to reduce redundant code and to prevent future inconsistency. For example, suppose the *Save Dialog* box is relevant to several test cases, the tester can write a common action function to open *Save Dialog* that contains a `click()` on the *File* menu followed by another `click()` on the *Save* item. On the other hand, testers can also define shared assertion functions to verify the same visual feedback that are derived from different actions. For example, the appearance of a *save* dialog box can be caused by a hotkey `Ctrl-S`, by a icon on the toolbar, or by the *Save* item in the *File* menu; all could be verified by `assertSaveDialog()`.
3. Testers can run a test script and monitor the progress as each test function in the script is run. They can see the summary showing whether each test has succeeded or failed as well as the total number of successes and failures.
4. When errors are found, testers can communicate the errors to programmers effectively. On the one hand, testers are encouraged to assign each test function a meaningful name, such as `test_click_play_button`. On the other hand, the images embedded in each function make it visually clear which GUI components and what visual feedback are involved in the errors.

Regression Testing

When a new feature is implemented, in addition to verifying whether the implementation is correct, it is equally important to ensure that it does not break any existing feature that used to be working. This practice is often known as *regression testing* in software engineering. Many software projects use daily builds to automatically check out and compile the latest development version from the version control system. The daily build is tested by automated unit testing suites to validate the basic functionality. However, because

of the weaknesses of automatic testing tools for GUI, current regression testing process is limited to work only on internal components but not on GUI. Therefore, regression testing becomes a tedious practice that requires QA testers to manually repeat the same set of tests whenever there is a modification to the GUI.

Sikuli Test is a labor-saving and time-saving tool enabling QA testers to automate regression testing. Using Sikuli Test, the testers only need to program test cases once and those test cases can be repeatedly applied to check the integrity of the GUI. To show the feasibility of Sikuli Test for supporting regression testing, an evaluation will be given later.

Test-Driven Development

While our testing framework is originally designed for QA testers, it can be used by both GUI designers and programmers during the development process. In large GUI projects where the separation between design and implementation is clearer, designers can create test cases based on design illustrations or high-fidelity prototypes. For example, a designer can use a graphic editor such as Photoshop to create a picture illustrating the GUI's desired visual appearance. Based on this picture, the designer can crop representative images of operable GUI components such as buttons to compose action statements. The designer can also graphically illustrate the expected visual feedback when these GUI components are operated. Again, this graphical illustration can be used directly in assertion statements. Test cases can be created and handed to programmers to implement the GUI's outward visual behavior. These test cases will initially fail because none of the desired visual behavior has been implemented yet. As more features are implemented, more test cases can be passed. When all the test cases are passed, the implementation is not only complete but also thoroughly tested. This practice is often known as *test-driven development*, which has been widely adopted by non-GUI development projects. Our visual testing framework initiates an opportunity for GUI designers and programmers to engage in this good practice of software engineering.

Even in small projects when a programmer often doubles as a designer and a tester, test-driven development can still be practiced. For example, given a design specification, a program can create the skin of a GUI without any functionality using a Rapid Application Development (RAD) tool. Then, before the actual implementation, the programmer can take the screenshots of the skin to write test cases and start writing GUI code to pass these test cases.

	appearance	disappearance	enabling	disabling	highlighting	unhighlighting	moved	text changed	shadow	focus	defocus	font changing	color changing	scaled-up	scaled-down	transparency	scrolled	toggle	text entered	text deleted	collapse	uncollapse	fade in	fade out	animation
checkbox	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
text box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
button	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
radio button	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list item	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree item	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree box	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
list	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2
tree	2	2	2	1	1	1	1	1	1	1	1	3	3	1	1	4	1	4	1	1	1	2	2	2	2

Table 1. The testability of GUI visual behavior under Sikuli Test.

EVALUATION

To evaluate Sikuli Test, we performed testability analysis—how diverse the visual behavior GUI testers can test automatically, and reusability analysis—how likely testers can reuse a test script as a GUI evolves.

Testability Analysis

We performed testability analysis on a diverse set of visual behavior. Each visual behavior can be defined as a pairing of a GUI widget and a visual effect rendered on it. We considered 27 common widgets (e.g., button, check box, slider, etc.) and 25 visual effects (e.g., appearance, highlight, focus, etc.). Out of the 675 possible pairings, we identified 368 to be valid, excluding those that are improbable (e.g., scrollbar + font changing). We began the analysis by applying Sikuli Test to test the visual behavior exhibited by four real GUI applications (i.e., 1: Capivara, 2: jEdit, 3: DrJava, and 4: System Preferences on Mac OS X).

Table 1 summarizes the result of the testability analysis. Each cell corresponds to a visual behavior. Out of 368 valid visual behaviors, 139 (indicated by the number of the application used to be tested) are *empirically testable*, visual behavior was found in the four applications and could be tested; 181 (indicated by a triangle Δ) are *theoretically testable*, visual behavior was not found in the four applications but could be inferred from the testability of other similar visual behavior; and 48 (indicated by an “F”) are *not testable*. In addition to these valid visual behaviors, there are 307 rarely paired improbable visual behaviors indicated by an “X”.

As can be seen, the majority of the valid visual behavior considered in this analysis can be tested by Sikuli Test. However, complex visual behavior such as those involving animations (i.e., fading, animation) are currently not testable, which is a topic for future work.

Reusability Analysis

We performed reusability analysis of test scripts based on two real GUI applications: Capivara, a file synchronization

tool, and jEdit, a rich-text editor. These two applications were selected from SourceForge.net with two criteria: it must have GUI, and it must have at least 5 major releases available for download.

First, we focused on the two earliest versions that can be downloaded of the two applications. For Capivara, we chose versions 0.5.1 (Apr. '05) and 0.6 (June '05) (Figure 5 A,B). For jEdit, we chose versions 2.3 (Mar. '00) and 2.41 (Apr. '00) (Figure 5 A,B). Since there were modifications to the user interface between these two versions, we were interested in whether test cases written for the first version can be applied to the second version to test the unmodified parts of the application. We created 10 and 13 test cases for Capivara and jEdit respectively. Most of the test cases were created using the test-by-demonstration tool, while some required manual adjustments such as giving hints and removing excess contexts from the detected visual feedback.

Table 2 summarizes our findings. These two tables include the first two versions, plus a later version that showed drastic change in the GUI for Capivara and jEdit respectively. The column of the first version shows how each test case is made: *A* denotes automatically generated, *AM* denotes automatically generated with some modifications, and *M* denotes manually written. Each column of the other two versions shows the result of each test case at the version: *P* denotes passed, whereas *F1* - *F5* denote failure. (The cause of each failure will be explained later.)

Between the first two versions of Capivara, we observed one modification: the size limitation of the panel splitter was different. Thus, we only needed to update 1 of the 10 original test cases to reflect this modification. In other words, we were able to apply the other 9 test cases against the second version to test the correctness of unmodified features. Similarly, in the case of jEdit, we observed 3 modifications among the features covered by the original 13 test cases. Again, we were able to apply the remaining 10 test cases against the second version.

Next, we examined the long-term reusability of test cases as the applications undergo multiple design changes. For Capivara, we considered two additional major versions: 0.7.0 (Aug. '05) and 0.8.0 (Sep. '06), whereas for jEdit, we considered five more: 2.5.1 (Jul. '00), 2.6final (Nov. '00), 3.0.1 (Jan. '01), 3.1 (Apr. '01), and 3.2.1 (Sep. '01). We tested whether each of the *original* test cases was still reusable to test the later versions and for those no longer reusable, identified the causes.

Figure 6 summarizes our findings. To show the reusability of the test cases, we arranged each version across the horizontal axis. For each version, the height of the baseline region (blue) indicates the number of the original test cases still being reusable for that version. This region exhibits a downward slope toward the direction of the newer versions, reflecting the fact that fewer and fewer of the original test cases remained applicable. The sharpest drop-off can be observed at version 0.8.0 for Capivara (Figure 5.c)

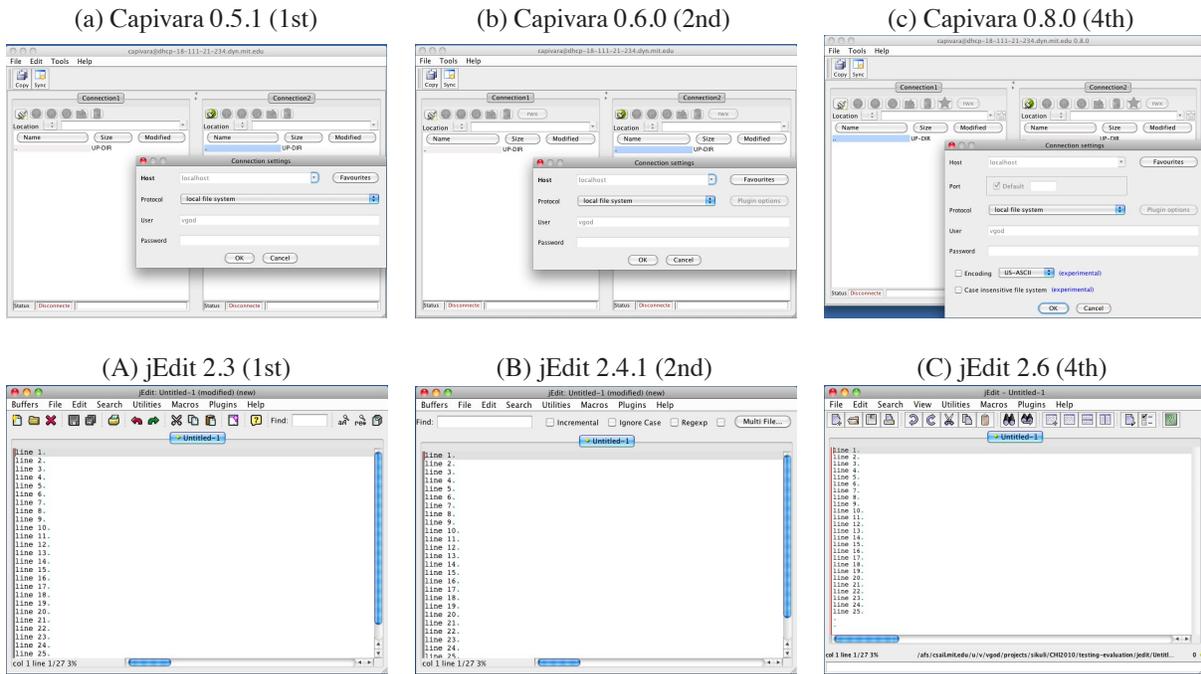


Figure 5. GUI applications used to evaluate the reusability of Sikuli Test scripts as these applications evolve.

and at 2.6final for jEdit (Figure 5.C), which can be attributed to the change of major design in these versions. The lesson that can be drawn from this observation is that as long as the design of a GUI evolve incrementally, as often the case, a significant number of test cases can be reusable, which is important for supporting regression testing.

Also, we identified five major causes for a test case to become unusable: (F1) change in the visual style, e.g. skin, size, font, etc.; (F2) removal of the action component; (F3) removal of the expected visual feedback; (F4) change in the surrounding of the target components; and (F5) change in internal behavior.

Each cause of test failures is represented in the figure as one of the colored regions above the baseline region, with its height indicating the number of unusable test cases attributed to it. As can be expected, the most dominant cause is change in visual style (F1, orange), since our testing framework is largely driven by high-level visual cues. One surprising observation is an unusual upward slope of F2 occurred at jEdit 2.5.1, indicating that test cases that were not reusable in the previous version became reusable. Upon close examination, we found that toolbar icons were removed at 2.4.1 but reintroduced at 2.5.1, making the test cases targeting toolbar icons reusable again. While such reversal of GUI design is rare in practice, when it does happen, Sikuli Test is able to capture it.

RELATED WORK

Testing is important for ensuring and improving GUI usability. Many research efforts have been made in the HCI community to study and develop tools for this critical task.

Boshernitsan et al [2] studied software developers' need to modify their programs to keep up with changing requirements and designs and developed a visual tool to simplify code transformation. Subrahmaniyan et al. [15] examined the testing and debugging strategies of end-user programmers and found testing and code-inspection are the two most common strategies. To support these strategies, Ko and Myers developed Whyline for the Alice platform [5] and further extended it to general Java GUI programming [6]. Using Whyline, a GUI programmer can make a visual recording of an interactive session with a GUI and *watch* this recording to catch incorrectly programmed visual feedback (testing). Whyline is able to intelligently suggest questions seeking to explain problematic visual feedback (e.g., *why did the color of this button change to blue?*). Such questions can help the programmer quickly lookup and fix the lines of code responsible for the problem (code-inspection). Sikuli Test can complement Whyline in that it can *watch* the recording on behalf of the programmer to catch errors and suggest appropriate questions.

A GUI needs to be interacted with in order to be tested. Several research works have focused on how to automate such interaction by demonstration, a paradigm known as Programming By Demonstration (PBD). As early as early 90's, Singh et al [12] proposed the Sage system that can capture and store GUI interactions demonstrated by users as reusable templates. Wilcox et al. [16] illustrated the value of visual feedback in programming by demonstration tools especially during the testing process, a finding validates the design decision of Sikuli Test to embed visual feedback directly in test scripts. Given the popularity of Web-based applications, the Koala system by Little et al. [8] and the CoScripter system by Leshed et al. [7] both aim to enable Web users to capture,

Test Cases of Capivara	(1st) 0.5.1	(2nd) 0.6.0	(4th) 0.8.0
connection-setting-cancel	A	P	P
connection-setting-ok	A	P	P
new-host-in-favorites	AM	P	F1
text-changed-in-status-and-tab	A	P	F1
menu-exit-dialog	AM	P	F2
toolbar-sync-dialog	A	P	P
name-size-column-in-listbox	A	P	F1
menu-options-tree	AM	P	F4
enabled-disabled-buttons	AM	P	F1
splitter-resize	M	F3	F3

Test Cases of jEdit	(1st) 2.3final	(2nd) 2.4.1	(4th) 2.6final
textarea-add-del-by-key	AM	P	F1
textarea-add-del-by-menu	AM	P	F1
new-tab-by-key	A	P	P
new-tab-by-menu	AM	P	P
new-tab-by-toolbar	AM	F2	F1
find-by-key	AM	P	F1
find-by-menu	AM	P	F1
find-by-toolbar	AM	P	F2
textfield-on-toolbar	AM	F5	F3
toolbar-print-dialog	A	F2	F1
menu-submenu	AM	P	P
scroll-textarea	M	P	F1
quit-cancel	A	P	F1

Table 2. Test cases created for the first version automatically (A), semi-automatically (AM) or manually (M) and their reusability (Pass or Fail) in subsequent versions (2nd and 4th).

share, automate, and personalize business processes. Zettlemoyer and St. Amant [19] developed VisMap for visual manipulation through an application’s GUI, which inferred a structured representation of interface objects from their appearance and passed the representation to controllers for manipulation. Based on VisMap, St. Amant et al. [14] then described several techniques of visual generalization for PBD, and enlightened the possibility of real-time screen analysis of screen images by PBD systems. In relation to these works, Sikuli Test extends PBD to serve a new purpose—GUI testing, and is also applicable to any Web-based GUI as long as its visual feedback is observable.

In the software engineering literature, many works have dealt with the issue of GUI testing from which many lessons and inspirations can be drawn. Xie and Memon [17] surveyed a large number of GUI testing frameworks and identified four common approaches: (1) writing scripts manually, (2) generating scripts by record-playback, (3) checking results with assertions, and (4) testing internal functionalities only. Sikuli Test supports the first three approaches by focusing on outward visual feedback. Memon [9] further pointed out that automation is important not only for simulating user interactions but also for verification of the results. To automate interactions, Kasik and George [4] built a tool that uses genetic algorithms to automatically generate test scripts to act like novice-user in an unpredictable yet controlled manner. Ostrand et al [11] developed a tool that can capture and replay interactions demonstrated by testers. This tool represents captured actions in a flow-chart that can be edited and rearranged. Similarly, Sikuli Test represents captured interactions as visual action statements that can be edited at

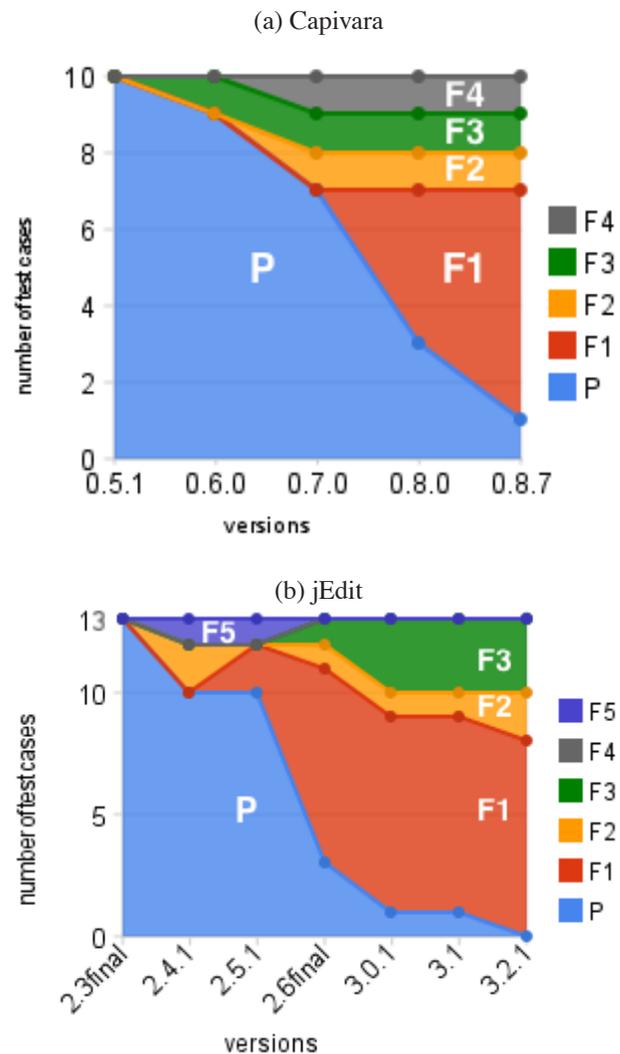


Figure 6. Long-term regression testing.

will. To automate verification, Memon et al. [10] developed the GUI Ripper tool that can explore a GUI and extract internal properties of GUI components to generate assertion statements meant to check those properties. However, most existing tools similar to GUI Ripper check results by inspecting the internal properties of GUI components through platform-specific APIs (e.g., Java APIs). Sikuli Test seeks to eliminate this platform dependency by inspecting the visual appearance directly.

Commercial GUI testing tools are also available to help QA testers perform their tasks, such as WinRunner [3]. However, most tools only allow the testers to script user interaction and direct the interaction by absolute positions. There is still a need for QA testers to verify the outcome of the interaction manually and to modify test scripts whenever certain GUI components are repositioned due to design changes. Sikuli Test seeks to eliminate this need by using visual cues to automate both the interaction and verification. CAPBAK

[13] is a rare exception of a tool that seeks to automate visual verification. It captures the image of the entire GUI after each user activity so that this image can be used later to test if the GUI still look exactly the same as before (i.e., regression testing). However, if the test fails, CAPBAK is unable to localize the error. In contrast, by detecting the visual changes before and after each activity, Sikuli Test can help the testers to pinpoint the problematic visual effect that requires the programmer's attention.

SUMMARY AND CONCLUSION

We presented Sikuli Test, a new approach to GUI testing using computer vision. Besides meeting the five design goals identified in an interview study with GUI testers, Sikuli Test offers three additional advantages:

1. **Readability of test cases:** The semantic gap between the test scripts and the test tasks automated by the scripts is small. It is easy to read a test script and understand what GUI feature the script is designed to test.
2. **Platform independence:** Regardless the platform a GUI application is developed on, Sikuli Test can be used to test the GUI's visual feedback. We have shown the examples of test scripts written to test traditional desktop GUI applications on Windows and Mac OS X, as well as Web applications in a browser and mobile applications in an Android emulator. Even though Sikuli Test is not designed to let users write scripts once and use them across multiple platforms, it is still possible to do so as long as the appearance of the applications looks the same.
3. **Separation of design and implementation:** Test cases can be generated by designers and handed to programmers to implement features that must pass the test cases, to eliminate the biases that may arise when programmers are asked to test their own implementation.

However, Sikuli Test currently has two major limitations that can be improved upon in the future. First, while Sikuli Test can assert what visual feedback is expected to appear or to disappear, it is unable to detect unexpected visual feedback. For example, if a programmer accidentally places a random image in a blank area, it is an undetectable error since no one would have anticipated the need to test that area with assertions. One solution would be to run the visual feedback detector at the test time to see if there is any detected visual feedback not covered by an assertion statement. Second, Sikuli Test is designed to test a GUI's outward visual feedback and is thus unable to test the GUI's internal functionalities. For example, while Sikuli Test can check if a visual feedback is correctly provided to the user who clicks the save button, it does not know if the file is indeed saved. One solution would be to treat Sikuli Test not as a replacement of but a complement to an existing testing tool. Together they make sure both the outward feedback and inward functionalities of a GUI can be sufficiently tested, a task neither can accomplish alone.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and UID group for great suggestions and feedback. This work was supported in part

by the National Science Foundation under award number IIS-0447800 and by Quanta Computer as part of the TParty project. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

1. Autoit. <http://www.autoitscript.com/autoit3/>, 1999.
2. M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07*, pages 567–576, New York, NY, USA, 2007. ACM.
3. HP Mercury Interactive. Winrunner. <http://www.winrunner.com>.
4. D. J. Kasik and H. G. George. Toward automatic generation of novice user test scripts. In *CHI '96*, pages 244–251, New York, NY, USA, 1996. ACM.
5. A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI '04*, pages 151–158, New York, NY, USA, 2004. ACM.
6. A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *CHI '09*, pages 1569–1578, New York, NY, USA, 2009. ACM.
7. G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripiter: automating & sharing how-to knowledge in the enterprise. In *CHI '08*, pages 1719–1728, New York, NY, USA, 2008. ACM.
8. G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *CHI '07*, pages 943–946, New York, NY, USA, 2007. ACM.
9. A. Memon. GUI testing: pitfalls and process. *Computer*, 35(8):87–88, Aug 2002.
10. A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269, Nov. 2003.
11. T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. *SIGSOFT Softw. Eng. Notes*, 23(2):82–92, 1998.
12. G. Singh and Z. Cuie. Sage: creating reusable, modularized interactive behaviors by demonstration. In *CHI '94*, pages 297–298, New York, NY, USA, 1994. ACM.
13. SOFTWARE RESEARCH INC. Capbak. <http://soft.com>, 1999.
14. R. St. Amant, H. Lieberman, R. Potter, and L. Zettlemoyer. Programming by example: visual generalization in programming by example. *Commun. ACM*, 43(3):107–114, 2000.
15. N. Subrahmanian, L. Beckwith, V. Grigoreanu, M. Burnett, S. Wiedenbeck, V. Narayanan, K. Bucht, R. Drummond, and X. Fern. Testing vs. code inspection vs. what else?: male and female end users' debugging strategies. In *CHI '08*, pages 617–626, New York, NY, USA, 2008. ACM.
16. E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *CHI '97*, pages 258–265, New York, NY, USA, 1997. ACM.
17. Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.
18. T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using GUI screenshots for search and automation. In *UIST '09*, pages 183–192. ACM, 2009.
19. L. S. Zettlemoyer and R. St. Amant. A visual medium for programmatic control of interactive applications. In *CHI '99*, pages 199–206, New York, NY, USA, 1999. ACM.