Image-Driven Automated End-to-End Testing for Mobile Applications

Caleb Fritz
Image-Driven Automated End-to-End Testing for Mobile Applications

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering

College of Engineering

University of Arkansas

Fayetteville, AR

By

Caleb Fritz
# Table of Contents

Abstract ........................................................................................................................................... 1

1. Introduction .................................................................................................................................. 2

2. Background.................................................................................................................................... 3
   Automated Mobile End-to-End Testing .......................................................................................... 3
   Image Detection Techniques ........................................................................................................ 5
   Two Popular Feature Detectors: SIFT and SURF ......................................................................... 7
   Finding and Filtering Matches ...................................................................................................... 9

3. Application Design and Implementation....................................................................................... 11
   Project Dependencies .................................................................................................................. 11
   Mobile App Test Subject ............................................................................................................. 12
   Testing Script Structure ............................................................................................................. 14
      Using Templates ....................................................................................................................... 14
      Writing Test Scripts ............................................................................................................... 16
   Testing Framework ..................................................................................................................... 17
      Entry Point ............................................................................................................................... 17
      Core and Activity Functions .................................................................................................... 20
      CV Functions ............................................................................................................................ 23

4. Experimental Approach .............................................................................................................. 25
   Test Script Creation .................................................................................................................... 26
   Device Simulator Setup ............................................................................................................. 31

5. Experimental Results and Analysis ............................................................................................. 32
   iOS Test Results ......................................................................................................................... 32
   Android Test Results ................................................................................................................. 36
   Testing Running Time ............................................................................................................... 40

6. Conclusions .................................................................................................................................. 41
   Future Work ............................................................................................................................... 43

7. References ..................................................................................................................................... 44
Abstract

The increasing complexity and demand of software systems and the greater availability of test automation software is quickly rendering manual end-to-end (E2E) testing techniques for mobile platforms obsolete. This research seeks to explore the potential increase in automated test efficacy and maintainability through the use of computer vision algorithms when applied with Appium, a leading cross-platform mobile test automation framework. A testing framework written in a Node.js environment was created to support the development of E2E test scripts that examine and report the functional capabilities of a mobile test app. The test framework provides a suite of functions that connect with an Appium server and provide interaction with the mobile test app to perform actions and assertions like clicking and verifying text. To do this without modifying the test app source code, the system employs image templates representing specific app components and identifies them within the test app by utilizing feature detection, matching, and filtering. From experimentation on three test scripts across multiple iOS and Android device simulators, iOS test script runs had a pass rate of 38% on average, while Android test runs had a pass rate of 74.5% on average. The test scripts ran perfectly only on the device simulators from which the template images were extracted via screenshots, while failures were mostly due to invalid or mismatched templates. Therefore, more generic templates that appeal to a variety of different device renderings are necessary for the test framework to be completely reliable.
1. Introduction

During this research project, we wanted to determine if computer vision techniques could be applied to current mobile end-to-end testing technology to improve upon it and create less invasive tests with equal or greater efficacy. A terminal-based desktop application was developed in a NodeJS environment and employed the testing library Jest to aid in the development of the various testing scripts, along with Appium, a cross-platform communication server used to send commands and receive feedback from mobile device emulators. The computer vision aspect of the project was implemented using a JavaScript binding of the OpenCV library, which exposed many different useful algorithms that would aid in determining the best techniques to apply for this research.

Three test scripts were written to examine different aspects of a UAConnect themed mobile application developed in a previous software engineering project. A login test script was written to test the aspects of the login system in the app, which involves detecting the University of Arkansas logo, entering and deleting text from username and password fields, and determining if upon entering correct credentials the user is logged into the app. A second homescreen test script was developed to test the navigational aspects of the app, which involves clicking on quick links and tabs before verifying the screen was navigated correctly. A third script involves testing a course planner feature that allows the user to view, create, edit, and delete courses from a vertically scrollable course list. These test scripts were then run on several different device simulators across different iOS and Android versions to determine if the testing framework developed serves as a reliable solution for mobile automated E2E testing.
2. Background

Automated Mobile End-to-End Testing

There are many different varieties of software process models that help tackle the challenge of creating effective organizational techniques that help produce a viable and maintainable software solution. Though these models have differing paradigms on project organization, one aspect remains constant – testing. Without the capacity to test a software solution, the understanding of its efficacy and efficiency in the real world is left unknown. There are many different forms of testing with a granularity that extends from testing individual logical branches and functions up to full-fledged system tests that include all moving parts of the solution. End-to-End testing, or E2E, is a higher-level form of testing that involves determining if a particular workflow from a starting to an ending point meets acceptable testing criteria within the software system. E2E can include some or even all aspects of a software system depending on if the workflow being tested requires those aspects to function. Fortunately, options exist today that aid developers in integrating E2E testing into their mobile applications.

Within the scope of mobile application E2E testing, a popular open source solution known as Appium provides cross-platform interactions with target applications on prevalent mobile OSes including iOS and Android. Appium is a standalone server application that exposes a suite of commands from an API that custom clients can interact with. There are a variety of client frameworks to work with for mobile, the most popular being WD.js and Webdriver.io, which both allow script writers to access commands in a NodeJS environment to interact with the Appium server (Intro. To Appium, n.d.). In order to write tests, special strings called locators must be placed on individual components within the target application in order to identify them once the test script is run. In the event locators cannot be applied to components, Appium
automatically generates locators called XPaths for each component and organizes them in a Document Object Model (DOM) structure as shown in Figure 1.

![Image of Appium Inspector utility and auto-generated DOM](image)

**Figure 1** – The Appium Inspector utility and auto-generated DOM with selected element properties.

However, several pain points exist in this implementation, resulting in test flakiness and workarounds that require expanding the client interface beyond its intended use in order to solve particular testing issues. Test flakiness can occur if locators change in the codebase, and in a large codebase that is interacted with often by many developers, locators may be changed accidently or even deleted, causing false positives when test scripts that use those locators are activated. These locators can also only be placed on custom components and may not be guaranteed to work for third party applications that do not explicitly support them. XPaths may remedy this for a time, but since they are automatically generated, adding new components will cause updates to the DOM, changing the XPath and causing test failures. This is when workarounds and expansion of the client API is necessary to solve these problems, resulting in less maintainable code. It is also difficult to test the look and feel of an interface using the
locator strategy, since locators can only be used to identify basic information like text content and the state that component may be in, rather than being able to test the aesthetic aspects of the component as well.

This project seeks to address some of these pain points through the use of computer vision techniques to provide visual locators that do not depend on editing the source code of a target application. These visual locators will allow for the dynamic identification of components, their contents, and their aesthetic aspects through which a suite of custom actions can be performed to test them in a variety of ways. Utilizing the vast API provided by OpenCV through a JavaScript binding, different computer vision techniques were investigated and integrated with current E2E testing technologies like Appium and WD.js in a standalone Node.js application.

Figure 2 – Template (left), source (mid), and pattern correlation (right) images, where brightness of pixels in pattern correlation image implies highest match (Perveen et al, 2013).

Image Detection Techniques

Originally, the template matching technique was investigated to determine if it would be the most effective candidate for implementation in this project. Template matching involves sliding a smaller template over equally sized patches of a larger background image and
measuring the similarity of pixel intensities between the template and each patch (see Figure 2). The template image is placed at the top left corner of the source image, and for each comparison, the template image slides one pixel from left to right before jumping to the next row of pixels until every pixel patch has been compared. Once the comparisons are complete, depending on the comparison technique used, the best candidate for a match will have either the lowest or the highest value, and its location can be extracted to determine the best matching patch (Perveen et al., 2013). While highly effective when searching for more distinguishable templates on source images, a successful search relies on matching a static feature of a particular size and orientation relative to the template image, making it less detectable if the source image is rescaled or reoriented. Template matching also requires the full template to be visible on the source image for a successful match and is unable to employ inference on partial matches. A possible solution to this problem involves iteratively adjusting the size and orientation of the template image and performing the template matching, however this is computationally expensive and less dynamic than another popular image detection technique: feature detection and matching.

Feature detection and matching involves detecting specific locations within images, called keypoint features or interest points, which are chosen to represent the image so that it may be matched against other candidate images that may be similar. Once keypoints are detected within an image, the region around each keypoint is supplied with a descriptor that represents a more salient and invariant description of the keypoint (Szeliski, 2010). The invariance of the descriptor is what makes feature detection and matching more powerful than template matching. Regardless of the scale and orientation of the template image, if it exists within a source image, the descriptors between them are capable of being matched, and the template image may be found within the source image even if it is partially occluded. This occurs when the descriptors
are matched against each other, often using a nearest-neighbor approximation, which is used in this research. An example of matching features is shown in Figure 3. Feature detection allows keypoints to be matched even in the presence of noisier images and is used in a wide variety of applications, including image stitching for panoramas and object detection.

Figure 3 – Descriptor matches between a template image (left) and source image (right).

**Two Popular Feature Detectors: SIFT and SURF**

The Scale Invariant Feature Transform (SIFT) approach was one of the first and most successful methods in detecting distinct features in images (Lowe, 2004). The first step of the method, called scale-space extrema detection, involves searching over all scales and image locations to determine scale and orientation invariant keypoints using a difference-of-Gaussian (DoG) function. The DoG function enhances edges in noisy images by subtracting a blurred version of an original grayscale image from another, less blurry version of that original image.
(Davidson, 2016). The second step involves keypoint localization, where at each candidate keypoint location, a model is used to determine location and scale, and the keypoint is selected based on its stability from the model. Next, orientations are assigned to each keypoint based on the local image gradient directions, and finally, those local image gradients are used to generate a descriptor for each keypoint that is invariant to distortions from shape and change in illumination. SIFT has a high accuracy when detecting keypoints but is computationally expensive and is less suited for real-time applications.

The Speeded Up Robust Features (SURF) method attempts to mitigate the computational complexity of SIFT while maintaining its accuracy by using approximation techniques (Bay & Tuytelaars, 2006). SURF uses box filters to approximate DoG calculations, citing that factors such as aliasing can reduce the accuracy of the calculations even when Gaussian filters are applied, thus an approximation better suited. This greatly improves the speed of the algorithm when using integral images, which gives a quick way of calculating the sum of pixel color intensities in an image (Mathworks, n.d.). The SURF descriptor calculations have also been reduced in complexity compared to SIFT’s implementation by applying an orientation to each keypoint based on information from the area around it, and then using a square region aligned to the orientation to extract the descriptor. There is also an even faster implementation of the algorithm that disregards orientation information for applications dealing with horizontal images. SURF’s ability to improve the speed of the SIFT algorithm while approximately maintaining its reliability and accuracy made it a prime candidate for this research and was used in the project code.
Finding and Filtering Matches

Once descriptors have been calculated using SURF, the descriptors on the template image must be matched against similar descriptors on the source image. Typically, a nearest-neighbor approach is taken to discover which descriptors match. This approach involves comparing the similarity of one descriptor in the template against multiple candidate descriptors in the source image. This similarity value is calculated using the Euclidean distance between the captured pixel intensities within a descriptor. For this research, the Fast Library for Approximate Nearest Neighbor Search (FLANN) is used to return the best possible matches using a collection of optimized algorithms for fast nearest neighbor search. The library requires a user to only provide the raw descriptor dataset, as it is capable of automatically determining the most effective approximation algorithm for returning the best matches using efficient data structures (Muja & Lowe, 2009). Though the resulting matches from FLANN find the best matches, some may still be outliers that match with descriptors that do not describe the object in the template image, thus another solution is needed to distinguish inliers from outliers.

The Random Sample Consensus (RANSAC) algorithm is applied to eliminate outliers from the collection of best matches (Fisher & Bolles, 1981). Since we wish to discover the four best points that correspond to the corners of the template image within the source image, RANSAC requires at least four matches. The algorithm then begins randomly selecting subsets of four matching point pairs and estimates a homography matrix (Camera Calibration, 2019). The 3x3 homography matrix represents the translation, scale, and orientation mappings between corresponding points in two images of objects that physically lie on the same plane, regardless of the camera perspective. Figure 4 shows how the homography matrix $H$ is used to map a template image point $(x_2, y_2)$ to a corresponding source image point $(x_1, y_1)$. 
Once the candidate homography matrix is generated, the number of matches that map within the bounds of the matrix are considered inliers, and the homography with the most inliers is considered the best possible mapping between the template and source image, allowing a bounding box to be generated as shown in Figure 5. Now that coordinate information about the template image has been contextualized within the source image, calculations necessary for performing actions like clicking and scrolling can be performed.
3. Application Design and Implementation

Project Dependencies

This project employs several software dependencies to aid in its development. These dependencies include the software packages WD.js, Jest, Tesseract.js, and OpenCV. WD.js is a popular Appium client API that allows interaction with the Appium server. Once the driver has been initialized with a capabilities object that describes the test environment, it connects to a listening Appium server. After initialization, a *driver* object is returned and exposes an API that allows a script writer to send commands to Appium, which will then perform those commands on a target mobile app and send back a response on the status of that action.

Jest is a testing library that provides a toolkit for organizing, executing, and verifying test scripts (Facebook Inc., n.d.). Jest was chosen as our test runner because of its powerful ability to run and show the output of tests in an intuitive way, making experimentation and testing easier (see Figure 6). Jest was initialized with several commands that provide a boundary on the amount of time a test is allowed to execute and force the testing session to close on the first failed test. Jest employs a recursive test searching strategy that allows users to enter in search criteria and run tests with a matching or partially matching name.

![Jest output from a passed test showing individual and total completion times.](image)

Figure 6 - Jest output from a passed test showing individual and total completion times.
Optical character recognition (OCR) was utilized through a JavaScript binding of Google’s open-source Tesseract OCR engine that applies machine learning to process and recognize text in over 100 languages, and it can also be trained to recognize other languages as well (Google Inc., 2019). For the scope of this project, localization for the target app remains in English, but the Tesseract Engine’s capabilities and ease of use made it an excellent choice for this project’s OCR needs.

OpenCV is a vast library of functions providing implementations of computer vision techniques, from low level image processing to high level face detection and feature matching (Pulli et al, 2012). OpenCV is an open source project and is maintained by OpenCV.org, a non-profit organization of developers, under the BSD license. Many different language bindings exist for the library, exposing its API to a variety of popular programming languages, including JavaScript. For a Node.js environment, the OpenCV binding “opencv4nodejs” acquired from npmjs.org was used in this project to utilize the necessary feature detection, matching, filtering, and image processing algorithms.

**Mobile App Test Subject**

Before beginning development on a testing application that could test mobile apps, a fully functional app with a variety of different capabilities found commonly across different apps is necessary in order to determine the efficacy of the testing application. Thankfully, a mobile app with such capabilities was created prior to the formulation of this research in a separate software engineering course. Common capabilities across mobile apps include several generic features like login screens and main navigation screens, along with actions like clicking, editing fields, and scrolling. This app proved to be an effective test subject, since it was created agnostically from the design of the testing framework and was not influenced or modified in any way to
better suit the testing framework. This provided more real-world testing situations and allowed a better understanding of the successes and failures of this research.

Figure 7 - (Left to right, top to bottom) Images of the login, home, profile, news, events, course planner, create course, and edit course screens.
The app, named UAConnect Mobile, provides users with the ability to login to a separate account to view university-related news and create or modify aspects of a personal profile along with course information as a means to plan for future semesters. The app stores user information within a real-time database provided by Google Firebase, which allowed for quick synchronization of user data between the backend and client when data was created, modified, and deleted. It was designed using the open-source, cross-platform React Native framework, giving us the ability to test the app using our framework on both iOS and Android mobile OSes. UAConnect Mobile provides a login screen, which upon successful authentication directs the user to a home screen with a bottom navigation bar, as well as four additional navigation buttons that can send them to separate screens that provide a certain feature. These features include profile management, viewing news and events, and course planning (see Figure 7).

Testing Script Structure

Using Templates

Rather than having to edit the source code of a target mobile app and inject custom labels into components so they may be identified by Appium, the test scripter will now use image templates to identify the components of the app they wish to test. For this project, screenshots of the target app were taken, and the images were cropped around the testable components in the image, producing templates. It is important for the script writer to treat templates of a component as a snapshot of a particular state of the component it represents. If the component’s state changes and mutates the way it is displayed, a new template should be taken for that component and used in further interactions. However, the system design allows for some flexibility in template reuse even if their state does change, like when editing an input field, which will be explained later.
Once the templates are captured, they are placed in a main template directory within the project code, and templates are organized further within directories that separate them by screens or features. For example, a template `emailInputField.png` representing the email input field on the login screen is placed within the `templates/screens/login` directory. Once a script is run, it automatically calls a function that performs a depth first search of the template directory and adds attributes to an initially empty object. The object’s attributes are modelled after the structure of the template directory, so if a directory is encountered, an attribute is created in the object, and that directory is expanded in the search, where further attributes are then added to that parent attribute until that directory path has been exhausted. This means all leaf nodes in the search tree are template image files, so once they are encountered, an attribute is created and given the absolute path of the image as shown in Figure 8.

**Template Object:**

```json
{ 
  "navigationBar": { 
    "eventsTab": "../templates/navigationBar/eventsTab.png",
    "homeTab": "../templates/navigationBar/homeTab.png",
    "newsTab": "../templates/navigationBar/newsTab.png",
    "plannerTab": "../templates/navigationBar/plannerTab.png",
    "profileTab": "../templates/navigationBar/profileTab.png"
  },
  "screens": { 
    "coursePlanner": { 
      "addCourseButton": "../templates/screens/coursePlanner/addCourseButton.png",
      "avgGPAEdited": "../templates/screens/coursePlanner/avgGPAEdited.png",
      "avgGPAInit": "../templates/screens/coursePlanner/avgGPAInit.png",
      ...
    }
  }
}
```

Figure 8 - Sample of the template object generated from DFS directory search routine.

Now that an object representing all the available templates has been created, it can now be used within test scripts to identify the templates and pass the absolute paths of those template images to a set of functions that perform actions and assertions on them.
Writing Test Scripts

A test script is a JavaScript file of the form `name.test.js`, where the name describes the feature or workflow it is testing. Every test script has some initial boilerplate setup, which involves importing resources like functions to start and end a session with Appium, testing functions, and the templates object. Jest provides an effective test script organization toolkit that script writers can use to organize their tests. This organization is through the use of `describe` and `it` functions. Both of these functions take a test name argument that describes the test step and a callback function that contains the content of the test step. The `describe` functions are often used to encapsulate many `it` functions, and possibly many `describe` functions as well.

```javascript
describe('Login Screen', () => {  
  const {  
    emailInputField,  
    passwordInputField,  
    mainLogo,  
    ...  
  } = templates.screens.login;

  it('should see the logo exists', async () => {  
    await Core.templateIsDisplayed(mainLogo);  
  });

  it('should be able to populate the email field', async () => {  
    await Core.imageHasText(emailInputField);  
    await Core.setText(emailInputField, 'Test@test.com');  
  });

  it('should be able to populate the password field', async () => {  
    await Core.imageHasText(passwordInputField);  
    await Core.setText(passwordInputField, 'fake-password');  
  });

  ...  
});
```

Figure 9 - Example of the describe and it function structure for the login test.
They typically represent a particular feature or workflow being tested, while the *it* functions test small sections and serve as the basic building blocks of the test script.

Every test script has a top level *describe* function that encapsulates the rest of the test organization. After the top-level *describe* is defined, all the templates needed for the test are extracted from the templates object and used throughout the rest of the test script. As shown in Figure 9, the templates are then passed to several different functions within a package called *core.js*, which then use the templates to calculate their positions in the target app and perform actions on them.

**Testing Framework**

**Entry Point**

The testing framework has two commands to start running tests either on iOS or Android platforms. These commands both take several parameters to dictate which test or tests should be run on what device simulator and OS version. By default, if no tests are specified then all tests are run under a *tests* directory within the project code, and default values for the simulator and OS version are used. Once a user decides on what platform to test on and parameters to supply, they must start an instance of the Appium server so that the WD.js client can connect to send commands and receive responses. For this project, Appium server versions 1.15.0 was used, which supports iOS versions higher than 9.0 and Android versions higher than 4.2. For greater efficacy, separate client instances for iOS and Android connect to their own Appium server instances on adjacent ports to avoid overloading a single server. This allows tests to be run on an iOS and Android device simulator simultaneously, increasing testing output. The final step before tests may be run involves compiling the test target UAConnect app so the testing framework can install instances of it on the simulators.
Once all desired user parameters have been gathered, the Jest command line interface (CLI) is initialized programmatically within the program entry point and is supplied with a variety of options as shown in Figure 10:

```javascript
const options = {
    ...(test && {_: [test]}),
    projects: [__dirname],
    silent: false,
    verbose: true,
    runInBand: true,
    setupFilesAfterEnv: ['<rootDir>/src/setupTestingFramework.js'],
};
```

Figure 10 - The options object used to initialize the Jest test runner.

The first property detects if tests have been specified and extracts them from the `test` information supplied by the user into an array. The `projects` property specifies the searchable domain where test files can be found. The `silent` and `verbose` properties force Jest to output the results of the test, providing the most information it can for a particular test run. The `runInBand` property forces all the tests extracted from the `test` parameter to be run in order, disallowing the option for multiple worker threads to be dispatched. While this option is useful for unit tests, only a single iOS and Android instance may be active at a time, thus only a single test may run on each one at a time. Finally, the `setupFilesAfterEnv` property gives the location to a file that will setup several Jest options that provide a testing timeout of 200 seconds and force the test run to shut down if a failure is detected. Once jest is started, it will find and run the test scripts specified, which activates a WD.js client that connects to the Appium server. The next step involves initializing the client with information provided by the user.
The capabilities object specifies the test environment, detailing the test target app and device.

Figure 11 - The capabilities object specifies the test environment, detailing the test target app and device.

The capabilities object shown in Figure 11 provides information about what environment the test will execute in, detailing the path of the target app’s packaged source code (either a file with a .app extension for iOS, or .apk for Android), the platform being used, and the device and OS version of the test simulator. Other attributes are platform specific and reduce memory usage or specify the app entry point. An instance of the client named driver is then initialized with the capabilities object, allowing it to connect to the listening Appium server. The driver is then exported so it may be used by a series of high-level functions that facilitate actions like clicking, verifying text, scrolling, and more.

```javascript
const capabilities = {
  ios: {
    app: "ios .app path",
    automationName: "XCUITest",
    deviceName: process.env.DEVICE ? process.env.DEVICE : "default iOS device",
    platformName: "iOS",
    platformVersion: process.env.OS_VERSION ? process.env.OS_VERSION : "default OS",
    preventWDAAttachments: true
  },
  android: {
    app: "android .apk path",
    appWaitActivity: ".MainActivity",
    automationName: "UiAutomator2",
    deviceName: process.env.DEVICE ? process.env.DEVICE : "default Android device",
    platformName: "Android",
    platformVersion: process.env.OS_VERSION ? process.env.OS_VERSION : "default OS"
  }
};
```
Core and Activity Functions

The client *driver* object, once initialized, can be used to issue low-level commands to the Appium server, such as clicking at a specific location, dragging across a screen between two boundaries, bringing focus to and activating the keyboard on an input field, and more. This functionality was abstracted into higher-level “core” functions to which the user would pass template image paths where they would be processed to fulfill generic actions such as clicking and scrolling. The core functions exist within a module that keeps track of the current test state through a `context` object shown in Figure 12:

```javascript
const context = {
  templateImagePath: string,
  sourcePolygonPoints: array of 4 point objects,
  sourceImageBuffer: Buffer,
  windowSize: object of width, height attributes
};
```

Figure 12 - The context object used to identify the current template being tested.

The last template image path passed into a core function is stored within `templateImagePath`, as well as point coordinates `sourcePolygonPoints` that identify the corners of its bounding box within a source image. A buffer containing the source image is stored in `sourceImageBuffer` and the reported size of the clickable region within the app is stored in `windowSize`. The `context` object serves to reduce the computation time by allowing the reuse of previous calculations on templates. Usually once a template is discovered, multiple actions are performed on that template, so rather than perform redundant calculations, the `context` object is overwritten with the results of the first calculations performed on that template, and any other core functions called with that template will use the same information. While more efficient, this implementation makes the critical assumption that templates represent particular states of the
component being tested, so if a new state of the component must be tested, a new template should be used, otherwise unexpected results could occur from the usage of old template state.

If a new template is being introduced to a core function, it must update the context object by calling a template searching function. All functions who encounter new templates call the findTemplate function outlined in Figure 13, which takes the template image path, an optional context image path, and an optional search timeout that is 10 seconds by default. The function begins by taking the current time in milliseconds and storing that in a startTime variable. The function will then loop until the timeout is reached. Within the loop, if the context.windowSize object is undefined, it will be set by making a call to a driver function through the helper function setWindowDimensions.

```javascript
function findTemplate(templateImagePath, contextImagePath, timeout = DEFAULT_TIMEOUT) {
  startTime = current time in ms
  while currentTime in ms - startTime < timeout
    if no windowSize context
      context.windowSize = setWindowDimensions()

  sourceImageBuffer = getCurrentScreen()

  if contextImagePath is found
    contextBoundingBox = getTemplateCorners(contextImagePath, sourceImageBuffer)

  if contextBoundingBox is found
    extract the image from the bounding box into contextImage
    templateBoundingBoxWithinContextImage = getTemplateCorners(templateImagePath, contextImage)
    templateBoundingBox = combineBoundaries(templateBoundingBox, contextBoundingBox)

    if templateBoundingBox is found
      save it to context and return it

  else no context was given, find the template within the current screenshot
    templateBoundingBox = getTemplateCorners(templateImagePath, sourceImageBuffer)

    if templateBoundingBox is found
      save it to context and return it
      sleep for 500 ms
    throw timeout error if template cannot be found
}
```

Figure 13 - Template search algorithm pseudocode.
An image buffer representing the current screen of the test app is then returned from the helper function `getCurrentScreen`. If `contextImagePath` is defined, then the user wishes to provide a context image that will be identified first within the source image by calling the `getTemplateCorners` function, which uses OpenCV feature matching and image processing tools. If the context template can be found, then the template image is found within the context image also using `getTemplateCorners`. If the call is successful, we receive an array of four point-objects representing the corners of a bounding box where the template is presumed to be within the context image, but because we want the context to be the original source image, we add the top left coordinate value of the context bounding box to all points of the template array to put it within the perspective of the source image instead. If no context image is provided, then the template image is searched for within the full screenshot source image. If the template image cannot be found, the search sleeps for 500 milliseconds, giving the target app time to update its screen before taking another screenshot and trying again. If the search exceeds the timeout, an error is thrown and the test run is terminated.

Many core functions were written to aid the user in performing actions on templates, and all allow the passing of context images if needed to further distinguish the template image. Beyond providing the user with the ability to click, verify text, and scroll using templates, they also have access to functions that can verify if templates do or don’t exist on the current screen and set or clear text from input fields. There are three utility functions available to the user, including a sleep function and functions that determine if the platform is running on iOS or Android. The sleep function pauses program execution for a specified number of milliseconds and can be used to provide more time for the target app to update its screen contents and reduce test flakiness. The platform detection functions are helpful when app behavior is different
between platforms while testing and require special testing steps for each one. Most core functions issue commands to the Appium server and capture possible errors to be passed up to the Jest test driver so the error can be reported, and the test can be shut down.

Some repetitive tasks like logging in and out require several core function calls, so to reduce redundant code, a separate group of functions called activities were created. The activity functions consist of login and logout routines, which are used at the beginning and end of each written test script, respectively. This allows users to provide login credentials to a single function call that abstracts several calls to core, making tests more readable. Core functions do not directly handle the calculations that generate the template bounding box and OCR results. Instead, they call functions from a lower-level module created to process templates and extract their bounding boxes or OCR results using the OpenCV and Tesseract libraries.

**CV Functions**

The cv module contains the logic for computer vision specific tasks including calculating and verifying template bounding boxes in a source image, extracting image masks from a particular region, and getting text from an image using OCR. The template bounding box is calculated in this module with the `getTemplateCorners` function, which was mentioned before as being used within the template searching function `findTemplate` housed within the core functions module. `getTemplateCorners` requires two parameters, being a template image and a source image, and will proceed to calculate the features on each of the images using the SURF algorithm provided from the OpenCV binding. The algorithm is initialized with a hessian threshold used to control the feature quantity to quality ratio, where a small threshold value results in more features of lower quality, while a larger threshold results in fewer but more salient features. The hessian threshold is set initially to 2500, which was found to produce a
sufficient number of features on both large and small templates through experimentation. Once the SURF algorithm calculates the features for each image, they are immediately passed into its descriptor function to calculate the descriptors for each feature. The descriptors in the template image are then matched against descriptors in the source image using the FLANN approximation library to get a set of the best matches between the two images. Once the best matches are discovered, the features from them are passed into the RANSAC algorithm to eliminate any outliers. At least four best matches are required to generate a homography matrix after they have been processed by RANSAC. The homography matrix is used within a perspective transformation function to map the original template onto the source image, returning an array of four objects representing the coordinates of the template corners within the source image. We now have a bounding box region of the template that can be passed back to the core functions, allowing them to perform actions on that region.

To determine if a bounding box is usable, several validation steps are necessary. First, the result of the homography matrix generation is checked. If a homography matrix could not be calculated, it will be empty, allowing the system to halt further calculations. Otherwise, once the candidate bounding box has been generated, it is passed to a validation function that checks if the four corner angles are at most plus or minus 15° from a right angle, enforcing that the bounding box be somewhat rectangular while allowing margin for error for less well-defined templates. Once the validation completes successfully, the bounding box is returned to the caller. In the event of an invalid homography matrix or bounding box, the hessian threshold is reduced by 50 and the process restarts with the recalculation of features in the template and source images. This loop will continue to decrement the hessian threshold while invalid results are still detected until it reaches the minimum threshold of 500. Once this occurs, the function returns null,
alerting the calling findTemplate function to acquire a new screenshot and restart the search process.

Another function called getImageFromRegion takes a source image and a bounding box as parameters and extracts the image formed by the bounding box within the source image. This function is useful for extracting a context image from a source image within the findTemplate function. It is also used to compare text between the template and its matching region in the source image to determine if the templates have the same text content. OCR is handled within a getImageText function that takes a template image path parameter and initializes the Tesseract engine to process and return the text within it. By default, if the engine does not have access to three resources, including paths to core files, a language file, and a worker thread file, it will download them automatically before proceeding. To avoid slowing down the engine with the download process, these files were manually installed into the project code and referenced to the engine so that they are used on every initialization. Tesseract provides a very simple interface that involves only spawning a worker thread and passing the image file to be processed through a recognize function call. If the worker returns with text, that text will be returned to the caller. Otherwise, an error will be thrown, and test execution will cease. In either case, the worker thread’s resources are released before the function terminates.

4. Experimental Approach

To test the efficacy of the framework, three test scripts were written and activated on the UAConnect app using a variety of different iOS and Android device simulators and OS versions, where each test script ran for three trials per simulator. All iOS template images were captured from an iPhone X simulator running iOS 11.4, while all Android template images were captured from a Nexus 6 simulator running Android 8.1.
Test Script Creation

The first test script tests several aspects of the app’s login screen, making sure that a user may edit the email and password fields, and upon passing in correct credentials, can press the sign in button to enter the app. Once within the app, the test determines if the login screen correctly navigated to the home screen by detecting the presence of the logout button in the header. Once the login button is detected, its text is verified to match with the text on the template, and it is pressed with the expectation that the app will navigate back to the login screen. Once this has happened, the email and password input fields have been cleared, and the script proceeds to test a login failure scenario. It simply presses the sign in button and searches for a popup error message. After the previous situations are met, the test shuts down and closes the session with Appium.

The second test focuses on aspects of the app’s home screen and its navigation capabilities to the screens housing main features, including the profile, course planner, events, and news screens. The script begins by calling the login activity with a test user’s credentials to enter to the home screen. To confirm the home screen has been navigated to correctly, the

Figure 14 - iOS (left) vs. Android (right) rendering of profile quicklink text (middle).

UAConnect title text above the navigation quicklinks has its text verified. For each of the features, their corresponding quicklink text is verified before being clicked. Once the quicklink
has been clicked and the feature’s screen is navigated, either a screen title or a defining feature of
the screen is sent to the templateIsDisplayed core function to confirm the screen’s existence
before clicking on the home screen tab on the bottom navigation bar, popping the screen back to
home. It is important to note an anomaly experienced during the development of this test script,
which involved differences in the way the templates rendered the profile quicklink text as shown
in Figure 14. iOS renders the dot of the “i” in “Profile” differently from Android, which caused
the OCR to fail on the platform opposite to the one the template was captured on. To solve this
problem, two templates were taken, and platform specific code was written to use the
corresponding template from the platform the test was executing on.

The final script tests the creation, modification, and deletion of a course card within the
course planner feature, verifying changes in text components along the way. It begins by
logging in and navigating to the course planner screen, where it will verify text in the screen
header, including the screen title and current values for the average GPA and total unit count.

Figure 15 - Initial states of the course planner screen (left) and create
course screen for iOS (middle) and Android (right). Notice the
difference in placeholder text between platforms.
calculated from all course cards in the scrollable list. Once the text in the header is verified, the “add course” button is clicked to begin filling out the course creation form (see Figure 15). Each field of the form has some initial placeholder text when the field is not populated. Unfortunately, the app does not display the same placeholder text on both platforms, forcing platform specific code and custom templates when editing each field within the test script. The course, term type, and term year fields bring up a scrollable list of items to pick.

The script searches for and chooses the “CSCE 491VH – Honors Thesis”, “Fall Semester”, and “2019” options from the course, term type, and term year lists, respectively (see Figure 16). These options then populate their respective fields in the form, and the test begins to fill out the gpa and units earned fields. The test first begins by finding and adding 3 to the units earned field before finding and adding 3.5 to the gpa earned field. The units field is edited first due to potential issues with the native keyboard blocking fields below the currently active field,
thus to avoid this issue the bottom field is edited first, keeping the native keyboard below all editable fields. Once the form has been filled out, the create course button is clicked, and a native confirmation modal shows with a cancel and confirm button. Since the modal confirmation dialogs are rendered differently between iOS and Android, platform specific code and templates were necessary here to allow the course to be created. Once confirmed, the navigation pops back to the main course planner list screen. The next step of the script’s execution involves scrolling down to the new card at the bottom of the list and checking that its contents are correct as shown in Figure 17.

All the information on the bottom course card is verified to confirm that the information supplied in the course creation screen has been added to the card correctly. Due to the presence of repeated components, the usage of context templates was very important for the success of
this test, otherwise components like the “Edit Course” button for the bottom course card would not be distinguishable from the others. After the course card is verified, its edit button is clicked, navigating the app to the edit course screen, where the number of units and the GPA will be increased to 4. After editing and submitting the card by clicking the save button and accepting the native confirmation modal, the card is instantaneously updated and navigation pops back to the previous scroll position in the course list as shown in Figure 18. At this point, the average GPA and total unit header information is then rechecked to make sure updates were made when the card was edited. The GPA and unit portions of the course card are checked to make sure they were updated as well.

The final script step involves clicking the edit button on the bottom card and deleting it. In doing so, the card should be removed from the list entirely, which is checked by calling the

Figure 18 - Edit course screen (left) and the resulting edited card (right).
templateIsNotDisplayed core function on a template of the course card title. Once the function returns, the script navigates the app back to the home screen where it presses the logout button and terminates. It is important to note that all of the test scripts use the sleep core function when animations are used in the app, like when the screen swipes to the left or right on navigation. This is done to avoid test flakiness caused from the matching of templates while they are in motion due to an animation.

<table>
<thead>
<tr>
<th>Mobile Platform</th>
<th>Device Simulator</th>
<th>OS Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOS</td>
<td>iPhone 6S</td>
<td>9.3</td>
</tr>
<tr>
<td>iOS</td>
<td>iPhone 7 Plus</td>
<td>10.3</td>
</tr>
<tr>
<td>iOS</td>
<td>iPhone 8</td>
<td>11.4</td>
</tr>
<tr>
<td>iOS</td>
<td>iPhone X</td>
<td>11.4</td>
</tr>
<tr>
<td>iOS</td>
<td>iPhone XR</td>
<td>12.1</td>
</tr>
<tr>
<td>Android</td>
<td>Nexus 5</td>
<td>6.0</td>
</tr>
<tr>
<td>Android</td>
<td>Nexus 6</td>
<td>8.1</td>
</tr>
<tr>
<td>Android</td>
<td>Pixel</td>
<td>7.1.1</td>
</tr>
<tr>
<td>Android</td>
<td>Pixel 2 XL</td>
<td>9.0</td>
</tr>
<tr>
<td>Android</td>
<td>Pixel 3XL</td>
<td>10.0</td>
</tr>
</tbody>
</table>

Table 1 - Test devices with OS versions.

Device Simulator Setup

iOS and Android device simulators each running different OS versions were created to test the UAConnect app using the written test scripts. Both iOS and Android were represented with five device simulators each that span across the latest versions of major OS version releases.
Table 1 shows the device simulators chosen with different resolutions and screen sizes to test the framework’s ability to work in a variety of different environments.

Due to OS limitations on simulators running within the scope of XCode 10.2, only major releases of iOS from 9 to 12 were available, thus the iPhone 8 and iPhone X were tested using iOS 11.4. Appium version 15 supported all simulators and OS versions except for the iPhone 6S running iOS 9.3, thus the server was downgraded to version 12 for that simulator only.

5. Experimental Results and Analysis

With ten test device simulators and three test scripts that are each run with three trials, a total of 90 testing sessions were recorded within a consistent testing environment where unnecessary processes were shut down to allow maximum CPU time for the device simulators. Each test was performed sequentially, where no simultaneous tests were run on iOS and Android simulators. Figures that show the percentage of a passing test represent the ratio of passing to failing *it* statements within the test script. At the end, data on the maximum average test time across devices for each platform is given and analyzed as well.

iOS Test Results

![iOS Login Tests](image)

Figure 19 - iOS login test results.
For the all iOS test runs, the iPhone 6S, 8, and XR simulators are unable to complete any part of the tests resulting in a test completion of 0% for each run, while the iPhone X performed well every time as shown in Figures 19, 20, and 21. The iPhone 7 Plus performed well for the login and home tests but faltered during the course planner test. Upon further inspection, it was discovered that the completely failing simulators produced severely inaccurate regions to
represent the locations for the email and password input fields. For more well-defined templates like the sign in button, the bounding box produced was also not ideal as shown in Figure 22.

![Figure 22 - iOS test failures due to invalid bounding boxes. Taken on the iPhone 6S simulator. Failures for iPhone 8 and XR are very similar.](image)

The failures here may be due to minute differences in how the UAConnect app is rendered on each device, causing the feature detection software to produce incorrect matches between templates, which for iOS were all captured from the iPhone X. Notice that across all test runs the iPhone X performs flawlessly, indicating that the system may be too sensitive to
differences in templates, thus requiring duplicated templates from each device to function properly.

The iPhone 7 Plus course planner trial failures occurred due to a popup success message that partially blocks the average GPA header in the course planner list screen (see Figure 23). The feature detection and matching algorithm’s ability to detect objects even when they are partially occluded allowed a rough bounding box for the header to be generated, but when the image was extracted and given to the Tesseract worker thread for OCR, it detected only the uncovered text. Upon returning the partial text back to the imageHasText core function, a mismatch was detected with the text on the template, and the test was terminated.

Figure 23 - iPhone 7 Plus course planner text mismatch error.
Because the iPhone 7 Plus test failures occurred due to unexpected results from the target application itself, the testing framework responded appropriately to this situation by failing, which would alert the test script writer to an unwanted situation within the app. Thus this particular set of failures should perhaps be treated as successful because they captured an unexpected and unwanted situation in the app. Using an Appium test id approach, since all components and their contents are stored within a DOM, the average GPA header text would be returned in full, and a situation like this would be overlooked by the testing framework.

**Android Test Results**

![Android Login Tests](image)

Figure 24 - Android login test results.
Figures 24, 25, and 26 represent the Android test results for the login, home and course planner tests, respectively. All Android simulators were able to successfully complete the login test for every trial however several issues began to materialize during the home and course planner tests for the Nexus 5, Pixel, and Pixel 3 XL, while the Nexus 6 and Pixel 2 XL remained successful.
We will focus first on the issues faced by the Nexus 5 test runs across the home and course planner tests. When run with the home test, upon logging in and verifying the text for course planner quicklink matches with what is found within the bounding box on the source image, the test shuts down with a text mismatch error, stating that the template text “CoursePlanner” does not match with the source text “CoursePlanner!”. Upon further inspection, it was discovered that the bounding box drawn around the quicklink text in the source image contains slightly different background content, which is perhaps resulting in the incorrect text detection for what is captured in the bounding box. A similar situation occurred during the course planner test, forcing test failures for the Nexus 5 trials as well. In that situation, the template text “CSCE491VH-HonorsThesis” is mismatched with “`CSCE491VH-HonorsThesis”, where an extra tick mark is added to the left of the text, which is likely an artifact of the bounding box capturing a sliver of the background on the left (see Figure 27).

Figure 27 - Nexus 5 course planner quicklink text bounding box (left) and create course options modal bounding box (right).
The Pixel also faces an issue interacting with a quicklink for all of its home test trials. Specifically, a valid bounding box cannot be generated around the profile quicklink so that it may be clicked. The template shows a quicklink of a slightly different aspect ratio than what is generated by the UAConnect app on the Pixel, potentially causing the failure to find the template within the app screenshot. For the course planner test trials on the Pixel, a mismatch occurs between the “Add Course” and “Edit Course” buttons shown in the course planner screen. The shared word “course” within both of these buttons probably contributes the most to their mismatch as shown in Figure 28.

The trial failures for the home test on the Pixel 3 XL were unfortunately due to an Appium window size calculation bug as shown in Figure 29. The size of the clickable region on the Pixel 3 XL excludes the bottom navigation bar of the UAConnect app, so any interactions with it will cause a test failure, since Appium commands to coordinates outside the clickable region will throw an error, terminating the test. A peculiar failure occurred during trial 2 of the
course planner test when a course card attribute with the text “UnitsN/A” could not be matched to the template text “Units3”. Unfortunately, the situation was not reproducible in the following trial to determine if the issue could happen consistently. It is unknown what caused this particular failure, but since the UAConnect application was being tested while connected to actual services, timing issues may have occurred that kept the data from being loaded in time.

**Testing Running Time**

![Testing Running Time Diagram](image-url)

**Figure 30 - Max average test pass time across devices.**
From the data for the average maximum running times of successful test trials shown in Figure 30, it was discovered that iOS simulators are significantly faster at completing tests over Android simulators. For the login test, iOS is able to complete approximately 21 seconds faster than Android. With the home tests, iOS completes nearly 33 seconds faster than Android (see Figure 30). The course planner test leaves the largest gap between them, where iOS completes approximately 133 seconds or 2 minutes and 13 seconds faster than Android. The difference in runtime is due to the architecture of Android simulators, which relies on a virtual machine layer beneath the Mobile OS and the target application, whereas iOS simulators are capable or running their code natively on the host CPU without an extra layer, giving iOS simulators a time advantage over Android simulators.

6. Conclusions

In this research, we implemented an E2E test automation framework that facilitates the creation of test scripts to explore the potential usage of computer vision techniques in aiding the development and maintenance of automated mobile E2E tests. Overall, iOS test script runs had a pass rate of 38% on average, while Android test runs had a pass rate of 74.5% on average. The average test pass rate for all devices running the login, home, and course planner tests is 70%, 48%, and 50.6%, respectively.

When we analyzed the results from the tests, it was discovered that the iPhone X, Nexus 6, and Pixel 2 XL simulators performed perfectly for all test trials. The image templates taken from the UAConnect app were captured from the iPhone X and Nexus 6, and as a result, the tests on those simulators were the most successful. All Android devices have a 100% test completion for their login test runs, indicating that the templates for the login screen were invariant to differences in screen resolutions across those devices. However, the Nexus 5 and Pixel began to
experience invalid or mismatched templates during the home and course planner tests. The Nexus 6 has a display with a resolution of 1440 x 2560 at 560 dpi, and since the Pixel 2 XL has the next closest resolution at 1440 x 2880 at 560 dpi, it is possible that the similar displays allowed the templates to work consistently on the Pixel 2 XL across all tests. However, devices of different resolutions completed the login test just as effectively but failed on the home and course planner tests, implying that the provided image templates are not generic enough to work consistency across devices. Unfortunately, the Pixel 3 XL was unable to complete all of the home test due to an Appium window size calculation error, but otherwise performed well with the exception of a unreproducible course planner test failure during its second trial run. The Pixel 3 XL’s results convey potential reliability concerns across certain devices due to Appium server issues, and may also be correlated with the UAConnect app’s performance during a live test with its services in use.

Interestingly, the iPhone 7 Plus passed completely on the login and home tests, only to fail after completing 68.5% of the course planner test due to the script’s failure to account for animated popup messages. This device’s display resolution is not at all comparable to the iPhone X display resolution, implying that differences in how each iOS device renders apps can cause invalid or mismatched templates, as was the case for the iPhone 6S, 8, and XR. Thus, as with Android devices, the image templates used for the tests were also not generic enough to work consistently across several devices. Due to differences in the Android simulator architecture, which runs on a virtual machine versus an iOS simulator that runs natively on a compatible CPU, it is expected that the Android simulator will perform slower, which the max average test time results corroborate.
Future Work

To improve the reliability of the template searching and matching, the usage of a Convolutional Neural Network may be necessary to better identify templates by assigning learnable weights and biases to candidate regions extracted from the test application screenshots. This enhancement may allow calculated features on candidate regions with slightly different renderings to be properly associated with the templates provided by the user, reducing the occurrence of invalid matches. This enhancement may also lead to the elimination of platform specific templates and code within the test scripts.

The amount of sleep operations due to animations within a test app could be heavily reduced as well by detecting template motion across each new frame taken from the test app. The extra sleep operations reduce test flakiness, but increase the runtime significantly, thus creating a more motion sensitive template searching algorithm will lead to faster test runs with a greater amount of protection against false positives.

Since physical devices were not available during experimentation, we were restricted to testing on device simulators, giving a less authentic understanding of the system’s performance. Thus, an important next step in testing would be to discover how the testing framework performs on physical devices to see if the results are similar.
7. References


