Image-based Approaches for Automating GUI Testing of Interactive Web-based Applications

Federico Macchi
University of Trento
Trento (TN), Italy
federico.macchi-1@studenti.unitn.it

Pierpaolo Rosin, Juan Marcos Mervi
Leonardo Company, Electronics Division
Ronchi dei Legionari (GO), Italy
pierpaolo.rosin, juanmarcos.mervi@leonardocompany.com

Luca Turchet
University of Trento
Trento (TN), Italy
luca.turchet@unitn.it

***Abstract*—Modern Graphical User Interface testing frameworks automate the testing of web-based interfaces to ensure the absence of functional and visual regressions. They employ common strategies to tackle different testing use-cases. This paper presents an analysis of these strategies, highlighting that it is not possible to test some kinds of interactive interfaces exhaustively. We propose a novel image-based framework that combines current techniques with new ones. These leverage Machine Learning and Computer Vision algorithms to analyze screenshots of the interface and prove its correctness. Results suggest that it suffices to automate the verification of interactive interfaces that were not fully testable before. Automated tests, developed as a benchmark, present almost no false-positives and high accuracy.**

I. Introduction

Nowadays, several millions of applications include a Graphical User Interface (GUI). In 2012, already more than 60% of the software under development was equipped with one [1] and it is reasonable to assume that this percentage is higher today. Since their introduction, GUIs evolved and increased in complexity along with the features of the software and reached a point where, in some cases, they are the only available method for the users to interact with or operate the whole software. In this case, ensuring that they work correctly is vital; functional and visual bugs can have serious consequences and severely degrade the usability of the system and the overall user experience.

In the field of Software Development, testing is defined as a technical investigation to assess the quality of the software or service tested and communicate it to the stakeholders [2], [3]. Testing approaches are divided into three groups; static, dynamic, and passive testing. In particular, dynamic testing is when the software is executed against a set of input values (for which the output is known), and its output is then compared to what is expected [4], [5].

Tests are further classified into four levels: unit, integration, system, and acceptance testing [3], [6], [7]. Unit testing is the testing of individual procedures within programs [5]; every method is tested independently from the others. Integration testing is performed on more software components at once and evaluates the interactions between them [8]. System Testing is conducted on fully integrated systems to evaluate its compliance with the specified requirements [5]. Finally, Acceptance Testing verifies that the developed system satisfies the stakeholders' needs.

GUI's testing is generally performed through Visual Testing, a type of dynamic testing, defined in the literature as a testing activity that verifies the presentation and functional properties of graphical elements under different conditions [1]. Presentation properties are visible features of the elements, such as color, shape, and dimensions; functional properties are all those characteristics not directly tied to the visual appearance of an element, such as its name, the supported actions, and the type of feedback communicated to the users [1]. For example, the presentation properties of a button that appears on a web page are its position, color, and shape; its functional properties are its name, unique id, and supported events.

This paper focuses on interactive web-based applications; their GUI is accessible through web browsers, supports multiple interaction methods, and its state can be altered automatically by the underlying system, without any action from the users. Elements present within these GUIs are rendered inside canvas elements, which are HTML elements used to create (and update) graphics at run-time [9].

There are currently many testing frameworks that address the problem of visual testing of web-based applications; the techniques they employ are similar and can be broadly divided into three main categories:

1) image comparison,
2) automation with WebDriver,
3) automation controlling users' input methods.

Image comparison is a technique used to discover visual regressions between releases; tests automatically compare interface screenshots of the new release with the previous one and report any difference found. Developers manually review the pairs and correct the issues if any.

This technique can catch subtle visual issues, that might be difficult to see for humans, because they perform a pixel-by-pixel comparison to determine if two images are equal or not. Basic image comparison algorithms are simple to implement, however, their scope of application is limited; interfaces with dynamic data must always use the same data, which might reduce the probability of catching a visual bug. Furthermore, this method does not verify user interactions. An example of a framework that uses this method is Percy.io [10].

The techniques belonging to the second category utilize WebDriver, an open-source tool for automated testing of web applications [11]; through WebDriver, developers can control the browser and its behavior, locate and access the

ISSN 2305-7254

elements present in the webpage, and inject custom JavaScript code to programmatically invoke functions. Developers can automatically verify all the functional properties of the web page elements; for example, they can ensure that all the CSS rules are applied, the text content is correct and, for interactive elements, the feedback is appropriate. This technique can be used to test atomic elements, but also asynchronous functions and animations; modern frameworks usually present APIs that allow developers to implement these verifications concisely.

However, techniques present in this category do not verify conditions and requirements visually; they rely on inspecting the HTML code to extract the information necessary for the verification. For this reason, they cannot ensure the correctness of the presentation properties nor can check the functional properties of all those elements whose code is not directly accessible, such as the ones based on the HTML canvas. An example of a framework that uses this method is Cypress.io [12].

The last technique analyzed, which belongs to the third category, uses Computer Vision algorithms to visually locate elements on the interface instead of WebDriver; in practice employs a similar approach of a human user, that scans the web page until s/he finds the desired element. To perform this search, a screenshot of the interface is saved (either in memory or on disk) and analyzed. The most common algorithm used to locate the element is Template Matching; given a template image, it slides it on the screenshot of the interface pixel-by-pixel until it finds a correspondence.

One of the frameworks that use this technique is SikuliX [13], which interacts with the elements found by controlling the users' input methods. This approach allows developers to test both the presentation and functional properties of elements since it does not rely on inspecting the code (which might not be always accessible) but can still interact with them. Furthermore, since it does not rely on WebDriver, SikuliX can automate virtually any application, including non-web-based ones.

Not having access to WebDriver, however, poses some problems; firstly, to perform di automation SikuliX needs exclusive control of the input methods and the screen of the users that are thus unable to perform other tasks. Secondly, it has been used successfully only on non-animated GUIs, a type of interface where there are no moving graphical components [14]; on animated GUIs, the method it uses to locate elements might fail. Indeed, the localization algorithm analyzes a still image of the interface; if in the meantime the element changes its location, the coordinates reported by the algorithm and the actual ones can be different.

From this analysis of the most used techniques, it emerged that there are still some gaps in the capabilities of current testing frameworks; some interactive interfaces cannot indeed be exhaustively tested. To address this issue, we present additional techniques that, combined with the current ones, aim to fill those gaps and increase the number of interfaces that can be tested.

II. Technology stack

To show the efficacy of the techniques proposed and provide concrete results in terms of testing time and accuracy, this paper presents them integrated with Selenium [15], a browser automation framework. In addition to Selenium, other technologies used are:

- Java is used to implement the algorithms,
- JUnit 5 [16] as the test runner to run the tests,
- Chrome Driver [11] to control Google Chrome,
- Jenkins [17] as the CI/CD system.

These components have been selected due to their popularity; according to the survey reported in [18], Selenium is often considered the de-facto standard for testing web-based software, and it is certainly the most popular; it also emerged that it is typically used in combination with Java (50%). In the Java ecosystem, JUnit is the most used test runner (75%). Finally, Jenkins is employed by most of the developers as a CI/CD system (72%).

Additionally, two well-established third-party libraries have been employed to implement the Computer Vision and Machine Learning algorithms: these are OpenCV [19] and Tesseract OCR [20], both used with their Java bindings.

III. Interface analyzed

The performance of the algorithms has been evaluated through some benchmarks on the 2D Tactical Map of Operator Stations of Flight Simulators. In Flight Simulators, the Operator Station interfaces instructors with the trainee (the pilot) [21]; as it can be seen in many commercial products [22]–[25], it usually comprises a map (referred to as Map or 2D Tactical Map from now on) and graphic pages, used to monitor the simulation and alter the parameters of the virtual environment or the simulated aircraft. The Map can be controlled by the instructor and also updates itself in near real-time to reflect changes in the simulation parameters and visualize actions performed by the pilot.

In particular, the Map considered in this paper (shown in Fig. 1) is produced by Leonardo Company Electronics Division and is used in their Flight Simulators; it displays the position of all simulation items, such as airports, tactical areas, and entities. In this context, an entity describes a military or civil unit present in the simulation and represented inside the Map; every entity is represented through unique symbols that follow precise military standards such as the APP-6A [26].

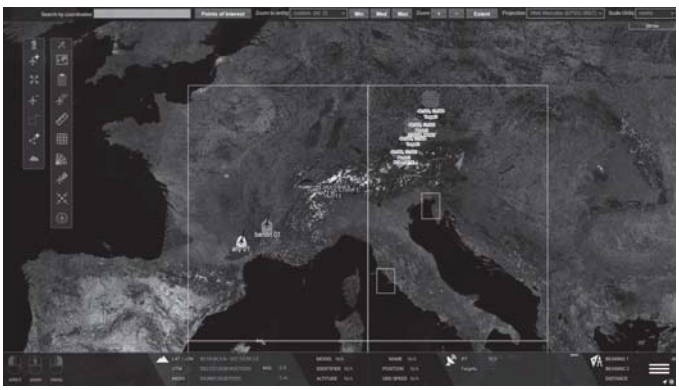

Fig. 1. Screenshot of the 2D Tactical Map analyzed

The Map offers to the instructors a quick overview of where objectives and other elements are located and basic tools to measure distances and control entities. Its core functionality is the visualization of data, that allows:

- merging geographical data with synthetic information to provide a very detailed geographical area representation,
- viewing simulation players as icons, providing related tactical information, and applicable quick actions that can be activated via keyboard shortcuts or more clicks;
- intuitively interacting with the system.

Its interface comprises multiple layers; the lowest one contains the map tiles, that are created inside an HTML canvas element. On top of it, all the entities are positioned in separate layers. All the map layers can be activated or deactivated by users with apposite toggles to show or hide additional information. Above the map layers, menus, and draggable widget bars are positioned; these contain tools that can be used by instructors during the simulation. Some of these tools allow instructors to track an entity (the Map follows an entity), display its trajectory, and send commands to it.

Apart from specific tools designed to help instructors during the exercises, general mapping tools are also available; for example, the Map can be panned, zoomed in or out, or at pre-defined resolutions, and it supports also markers and drawings to annotate specific areas.

In general, during a simulation exercise, the interactions instructors-map need to be quick and precise; instructors need to check often the simulation status and verify the behavior of the trainee and their feedback to planned and unplanned malfunctions or weather conditions. Sometimes, they need to impersonate an entity to show how a certain maneuver is performed; in this case, they can control the simulation using HOTAS (Hands-On Throttle And Stick) devices to pilot it. In general, however, the Map is controlled only through its Graphical User Interface with a mouse and a keyboard.

This interface has been chosen for the benchmarks because it respects all the properties defined before; it is accessible through a web-browser, it is interactive, and it updates itself to reflect changes in the simulation state or user actions. Furthermore, it is also animated; entities, for example, are moving graphical components.

IV. Algorithms and Use-cases

The techniques presented in this paper refer to three use-cases:

1) Pattern Detection,
2) Template Detection,
3) Text Recognition.

All of them implement image processing strategies to improve screenshots of the interface before performing the verification. This processing usually simplifies the work of the tests by eliminating irrelevant features and enhancing the ones needed, and, in some cases, optimizes the computation time by reducing the dimensions of the images.

A. Pattern Detection

The Pattern Detection use-case aims to identify patterns, such as lines and shapes. These patterns are distinguishable from other elements by their color, which is usually known to developers. This use-case is utilized when it is necessary to verify if something is being drawn on the Map; for example, if the trajectory of an entity is displayed in the correct colors and position (the trajectory color is the same color as the aircraft, and starts from its tail). Because patterns are usually abstract shapes, it is difficult to create a template that can be used for template matching; furthermore, common Computer Vision filters such as the Canny Filter or the Hough Transform (both the Standard and the Probabilistic) are useful for isolating and enhancing some shapes (for examples lines) but cannot be used to detect if they are present or not in the image automatically.

The algorithm automates the visual verification process by performing a pixel-by-pixel analysis on the screenshot. Initially, the pattern color is isolated from the others applying thresholding and masking techniques; for simpler cases, where the color is unique and different from the others present in the interface, the image is binarized. Colors different from the pattern one are set to black (or white, or grey depending on the starting color). In complex cases, where patterns might appear in random locations of the Map, its color is isolated using masking techniques such as the one proposed by D. J. Hemanth and U. Kose [27], which uses the HSV color scheme to isolate colors.

After the processing phase, through a linear scan of the image from the top-left corner to the bottom-right, the algorithm locates the first pixel of the pattern using its color information. Once the first pixel is found, the linear scan is stopped; if the shape of the pattern is known, the algorithm proceeds to select the next pixel and verifying if its color is correct until the whole pattern is found.

Listing 1 shows an example of how an horizontal straight line is detected in Java.

When the shape depends on non-controllable variables and is thus not known, the algorithm ensures that its pixels are connected and that the location of the pattern is correct. For example, to verify if the trajectory of an entity is being correctly drawn on the Map, the algorithm checks that its starting point is close to the tail of the entity, the trajectory is continuous (there are no holes in it), and that the ending point is reasonably far away from the starting one.

B. Template Detection

The Template Detection use-case aims to locate template images inside the interface; a template image represents a unique element of the interface and cannot have any variation in color, shape, or dimensions. It is used when there is the need to verify if an element appeared on the interface and retrieve its location to interact with it. In the case of the Map, it is mostly used to find map-related elements, such as markers and entities.

The algorithm uses the Template Matching algorithm provided by Open CV [28] combined with the image processing techniques described before. Template Matching is a robust algorithm that has been studied and employed in a lot of

Listing 1 Horizontal straight line detection algorithm.

```
int[] findFirstPixel(BufferedImage img) {
    int[] firstPixel = new int[2];
    for (int i = 0; i < img.getWidth(); i
    ↪  += 1) {
    for (int j = 0; j < img.getHeight(); j
    ↪  += 1) {
        // If the color of the current
        ↪  pixel is correct
        if (isSearchColor(img, i, j)) {
            firstPixel[0] = i;
            firstPixel[1] = j;
            return firstPixel;
        }
    }
    }
    return null;
}

int CountPixels(BufferedImage img) {
    // Get the location of the first pixel
    int[] firstPixelLocation =
    ↪  findFirstPixel(img);
    if (firstPixelLocation == null) {
        return 0;
    }
    int count = 0;
    int x = firstPixelLocation[0];
    int y = firstPixelLocation[1];
    while (x < img.getWidth()) {
        // If the color of the current
        ↪  pixel is correct
        if (isSearchColor(img, x, y)) {
            count += 1;
            x = x + 1;
        } else {
            return count;
        }
    }
    return count;
}
```

different scenarios, with positive results for static images and videos (see, for example, [29], [30], [31], [32]). The algorithm compares the template image with the source image by sliding it pixel-by-pixel; for each location, it calculates a score that defines how "good" is the match, which is stored in a matrix. Finally, using one of the available metrics it returns the location of the best match.

The implementation provided by Open CV does not take into account the cases when the template image has a transparent background. In this case, the algorithm performs poorly because when the template is positioned above the Map the background will not be constituted of transparent pixels anymore, but contains the terrain of the Map underneath; thus, when calculating the similarity score, all the background pixels will be different.

In general, the greater the background/foreground ratio is the lower will be the resulting similarity score. The results of the standard Template Matching algorithm show an average score of 70.46%, which is too low to avoid false positives (for example, SikuliX which uses this algorithm considers 70% as the minimum admissible score [33]).

To address this issue, before applying the Template Matching, the algorithm calculates and applies a mask, using a similar technique described in [27]. The mask aims to remove everything but the template from the screenshot to improve the detection score by removing foreign elements (such as the background pixels). The result is shown in Fig 2. It is calculated with the following steps:

1) the screenshot is converted to the HSV color space,
2) using lower and upper bounds provided by the developer, the mask is created; if the color of a pixel in the original image is within the defined range, its corresponding one in the mask is set to 1. Otherwise, it is set to 0. The final mask image has the same dimensions as the original screenshot and contains only 0s and 1s;
3) through the bitwise AND operator, the mask is applied; the AND operator multiplies the mask pixels with the original image ones. The result is an image where only the pixels that were set to 1 in the mask retained their original values and all the other ones are black (set to 0);
4) apply the standard template matching algorithm.

The code in Listing 2 illustrates the process described in Java.

Fig. 2. From left to right; the original screenshot of the interface, the calculated mask, and the final image with the mask applied. Note that the template is preserved without modifications by the mask, while the rest of the interface is hidden

C. Text Recognition

The Text Recognition use-case aims to extract textual elements from the interface and store them in string variables; in this way, they can be compared with their expected values to see if they are correct or not. It is useful when it is necessary to read a text to verify that a function is correct; for example, to test that the Map can be properly zoomed, the automated test reads the text that indicates the current map resolution, and compares it with the expected one, since to each zoom level is associated a specific resolution.

Text recognition capabilities are provided by Tesseract OCR. This library requires two main components: a Machine Learning model and an input image. For the Machine Learning model, a pre-trained one has been used (tessdata_best, which is one of the most accurate models [34]); since the fonts used are standard ones and the whole interface contains only English texts, there was no need to create an ad-hoc model. The input image is, as usual, provided by taking a screenshot of the interface.

Listing 2 Enhanced Template Matching algorithm.

```
void MatchHSV(Scalar lBound, Scalar
 ↪  uBound) {
    // Local variables
    Mat hsvSource = new Mat();
    Mat mask = new Mat();
    Mat maskedImage = new Mat();
    // Conversion to the HSV color space
    Imgproc.cvtColor(this.src, hsvSource,
     ↪  Imgproc.COLOR_RGB2HSV);
    // Mask evaluation
    Core.inRange(hsvSource, lBound,
     ↪  uBound, mask);
    // Bitwise and operator on the source
     ↪  image and the mask
    Core.bitwise_and(this.src, this.src,
     ↪  maskedImage, mask);
    // Perform the standard template
     ↪  matching algorithm on the masked
     ↪  image
    return this.matchTemplate(maskedImage,
     ↪  this.template, this.output,
     ↪  Imgproc.TM_CCORR_NORMED);
}
```

As for the previous use-case, the standard algorithm did not perform well; the initial results obtained by directly plugging in the screenshot of the Scale Line element showed an average accuracy of around 60%, which is not sufficient to avoid false positives. However, while previously the algorithm was not accounting for partly transparent template image, this time the input image is at fault; to improve the accuracy, many techniques described by the literature and the documentation of Tesseract OCR were applied.

1) Conversion to grayscale; [35] demonstrated that Tesseract OCR consistently performed better on grayscale images.
2) Cropping; cropping the image allowed to reduce to the minimum the presence of foreign objects that might interfere with the text extraction.
3) Resizing; the Tesseract documentation and the literature recommend the usage of images whose height is at least 20 pixels [36] and as close as possible to 100 pixels [37] to achieve the best results.
4) Unsharp Masking; this technique sharpens the input image to increase its quality. It is achieved by subtracting the smoothed image from the original one [37]. In particular, in Open CV this technique is realized through the addWeighted [38] function which is used to blend the blurred image and the original one.
5) Thresholding and inversion; the image is first converted into binary and then inverted to obtain a black text on a white background, as required by the Tesseract OCR documentation [36].
6) Morphological operations; opening and closing hats [39] are two morphological operations used to remove small objects (bright, on a dark background) and holes (darker region inside a bright object). They are used to further enhance the text before the extraction.
7) Text cleaning; finally. After the text has been extracted from the processed image, it is cleaned to remove all non-alphanumeric characters and whitespaces.

The code in Listing 3 illustrates the process described in Java; the values of the functions' parameters have been evaluated empirically and tuned to achieve the best results.

Fig. 3. Side by side comparison of the initial screenshot and the processed one, with their real dimensions

Listing 3 Post Processing steps.

```
Mat PostProcessImage(Mat src) {
    // Mat src is already in grayscale and
     ↪  cropped
    // 3. Resize
    if (src.height() < 100) {
        double scale = 4.6;
        Imgproc.resize(src, src, new
         ↪  Size(src.width() * scale,
         ↪  src.height() * scale),
         ↪  Imgproc.INTER_CUBIC);
    }
    // 4. Unsharp Masking
    Imgproc.GaussianBlur(src, src, new
     ↪  Size(1, 1), 3);
    Core.addWeighted(src, 1.5, src, -0.7,
     ↪  0, src);
    // 5. Thresholding and inversion
    Imgproc.threshold(src, src, 127, 255,
     ↪  Imgproc.THRESH_BINARY);
    Core.bitwise_not(src, src);
    // 6. Morphological operations
    Mat element =
     ↪  Imgproc.getStructuringElement(
        Imgproc.CV_SHAPE_RECT, new Size(1,
         ↪  1), new Point(0, 0));
    Imgproc.morphologyEx(src, src,
     ↪  Imgproc.MORPH_OPEN, element);
    Imgproc.morphologyEx(src, src,
     ↪  Imgproc.MORPH_CLOSE, element);
    return src;
}
```

V. How to write a test

Writing a test is straightforward; if using JUnit as a test runner, it is sufficient to create a function and annotate it with the @test annotation. In this way, JUnit will automatically execute it. Inside the test function, instantiate the class corresponding to the functionality desired; for example, to check if

an element is present in the interface, instantiate the Template Matching class. This class receives, as parameters, the template to use and a screenshot of the interface (which can be acquired with Selenium or other browser automation tools). Then, it is possible to call one of the available match methods, such as the MatchHSV one (its implementation is shown in Listing 2), that will perform the Template Matching algorithm and return an object containing the location of the template and the similarity score. The MatchHSV function requires two parameters (lower and upper bound) used to create the mask (shown in Figure 2) and isolate the template from the rest of the interface.

VI. Results

As mentioned before, to verify the efficacy of the techniques and the improvements concerning the standard algorithms provided by OpenCV and Tesseract OCR, a series of tests have been implemented: these tests automatically performed the GUI testing of some of the features present in the 2D Tactical Map. The results collected includes the average accuracy score and time to complete the test; the averages were calculated by running the test on the same machine and with the same conditions for 100 times. For the Template Matching, the average has been calculated by running it 100 times per each metric for both its standard and enhanced versions.

The Template Matching algorithm has been evaluated with the *CenterMapOnEntityTest*; this test verifies if the *CenterMapOnEntity* feature works correctly. This feature allows users to quickly center the map around any of the entities present in the Map during the simulation and it is useful to find an entity when there are multiple ones together. When the function is activated, a (green) marker (the one shown in Fig. 2) appears around the entity. The automated test invokes the function and takes a screenshot; then, it applies the Template Matching algorithm to verify that the marker appeared and is positioned in the center. Finally, if the marker is found, it creates an annotated image with the similarity score and bounding boxes around the marker location (See Fig. 4).

Fig. 4. Result of the *CenterMapOnEntityTest*

Tables I, II show the benchmark results of the Template Matching algorithm used by the test; the rows display the metrics used to calculate the similarity score and the execution time, while the columns contain the results of the standard and enhanced versions. From the tables, it emerges that the application of the masking technique consistently improved the results in terms of accuracy, without affecting too much the time needed to perform the validation (in the worst-case scenario, there is a 105 ms increment). Given the scores, the test employs the CCORR_NORMED metric used for the Template Matching.

TABLE I. Template Matching algorithm results (Time)

Metric	Standard	Enhanced	Variation
CCORR_NORMED	1173 ms	1278 ms	+8.95%
CCOEFF_NORMED	1237 ms	1278 ms	+3.31%
SQDIFF_NORMED	1177 ms	1210 ms	+2.8%

TABLE II. Template Matching algorithm results (Score)

Metric	Standard	Enhanced	Variation
CCORR_NORMED	0.7631	0.9880	+29.47%
CCOEFF_NORMED	0.8112	0.9851	+21.44%
SQDIFF_NORMED	0.5395	0.9757	+80.85%

The techniques belonging to the Text Extraction use-case were instead tested with the ZoomInTest; this test verifies that the Map can set its resolution correctly for all the zoom levels, starting from the lowest (level 2) to the highest one (level 22). As mentioned before, each zoom level is associated with a specific Map resolution; the test zooms in the Map, then reads the Scale Line element, which displays the current resolution, and compares its text with the expected one. Compared to the previous test, which verified a high-level feature, this one focuses on a lower-level one to check the zoom capabilities, that are used by many other functions.

The average processing time for the 20 zoom level is 3.668s, while the average score is 1.0, or 100%; note that this score was achieved by tailoring the techniques described before to this specific task. It is possible that to obtain similar results for other elements present in the interface a different combination of methods is needed; in particular, cropping and resizing the image might not be needed at all, and the values used to apply the unsharp masking method need to be calculated ad-hoc for the new element. The method that drastically improved the initial score was the image resizing and the unsharp masking which improved the final score by around 10% and 5% respectively. Even though the test processes 22 images for every run, the memory footprint of the algorithm is negligible; the largest generated image is 4KB.

The Pattern Detection techniques are tailored to verify very specific cases, where the Template Matching is not sufficient and the Text Extraction is not needed. These techniques are employed by pass-or-fail tests, that do not have a score; indeed the pattern is either found (or at least part of it) or not. Since the dimensions and shape of the pattern are not known in advance, it is not possible to calculate a score based on the ratio of pixels found/total pixels. The tests use a tolerance score, which defines the minimum number of pixels that must be discovered to consider the test as passing. For this reason, the score reported is the percentage of tests that passed out of 100, and not the average of each test run; the average time estimates the time needed to complete the algorithm. Compared to the other benchmark, the average time per run is longer because the test uses the Template Matching algorithm to locate the entities, and injects complex JavaScript code to activate the feature.

The test devised to verify the efficacy of the Pattern Detection algorithms is the TrajectoryTest; this test verifies

if the Trajectory feature works correctly. The purpose of this feature is to visualize the trajectory that an entity has traveled; the trajectory is represented with a line that starts from the tail of the entity and has the same color as it. The automated test invokes the relevant functions to activate the feature on a target entity, then waits for some seconds for the trajectory to appear (the trajectory is not drawn when the feature is inactive to save resources); after the time elapsed, it centers the map around the target entity and grabs a screenshot of the interface. The centering process is needed to ensure that the target entity appears in the screenshot (since it traveled in a certain direction for some time, it might have left the viewable area). Finally, the screenshot is processed to determine if the trajectory is present and if it starts close to the tail of the entity and has the correct color.

The result of this procedure is an annotated image that shows the starting and ending points of the trajectory and bounding boxes around the entities present on the Map, together with the detection accuracy, as shown in Fig. 5 below.

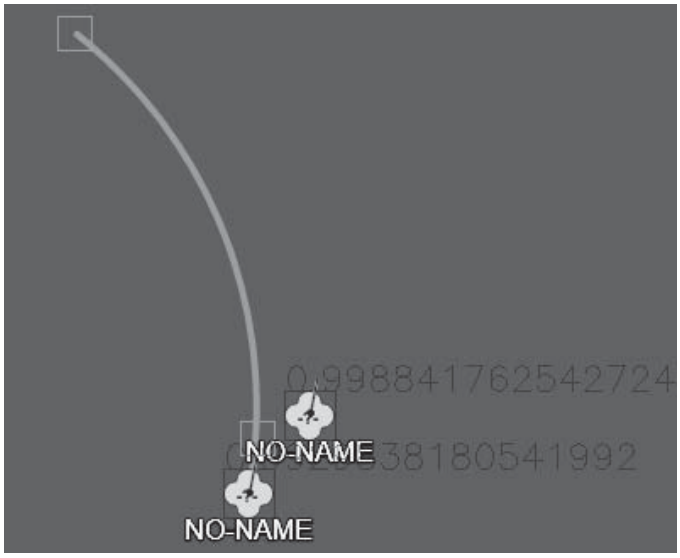

Fig. 5. Result of the *TrajectoryTest*

Table III shows the results collected; as expected, the average execution time is higher compared to the other tests due to the greater number of operations that the test performs. The score cannot be directly compared because, as mentioned before, it is not the accuracy of each test run, but the ratio of tests passed versus tests executed. Nonetheless, given the fact that the shape and dimensions of the pattern are not known in advance and that, since the trajectory is drawn at run-time, a small delay when acquiring the screenshot can affect the pattern, it can still be considered a good score.

TABLE III. PATTERN DETECTION ALGORITHMS RESULTS

Score (%)	Average execution time (ms)
74	7873

Compared to the standard version of the algorithms implemented in the OpenCV and TesseractOCR libraries, this paper presented enhanced versions, targeted at this particular interface; despite these enhancements, the time and space complexities of the algorithms did not change. So, to estimate the computational power and time required, it is possible to study directly the source code of the original algorithms, considering that the enhancements add a little overhead.

VII. CONCLUSION

This paper showcased techniques that aim to provide software developers and testers with additional GUI testing methods to automatically test interactive user interfaces, that cannot be fully tested using the current state-of-the-art frameworks. These techniques, categorized into use-cases, employ well-established and robust algorithms to analyze screenshots of the interface and verify the correctness of the functions' output.

Through Computer Vision and Machine Learning algorithm, they can check if elements appeared and their positions, extract and read texts, and discover abstract patterns. Thanks to the integration with WebDriver, they can inject JavaScript code to simulate user interactions and access low-level functions as well.

Finally, since they do not depend upon any specific pre-existing framework and do not have particular requirements, they can be easily integrated into CI/CD systems such as Jenkins and common test runners such as JUnit 5.

Techniques' efficacy has been proven through their extensive usage in automated tests performed the 2D Tactical Map, a complex interface packed with features; the preliminary results demonstrated that is possible to create automated tests to visually verify advanced and interactive user interfaces and ensure their correctness.

REFERENCES

[1] A. Issa, J. Sillito, and V. Garousi, "Visual testing of Graphical User Interfaces: An exploratory study towards systematic definitions and approaches," *2012 14th IEEE International Symposium on Web Systems Evolution (WSE)*, Sep. 2012.

[2] C. Kaner, *Exploratory Testing*, Nov. 2006. [Online]. Available: https://www.kaner.com/pdfs/ETatQAI.pdf

[3] W. E. Lewis, *Software Testing and Continuous Quality Improvement*. CRC Press, Jun. 2017.

[4] D. Graham, E. V. Veenendaal, and I. Evans, *Foundations of Software Testing: ISTQB Certification*. Cengage Learning EMEA, 2008.

[5] W. L. Oberkampf and C. J. Roy, *Verification and Validation in Scientific Computing*. Cambridge University Press, Oct. 2010.

[6] K. Wiegers, *Creating a Software Engineering Culture*. Pearson Education, Jul. 2013.

[7] J. A. Clapp, S. F. Stanten, W. W. Peng, D. R. Wallace, D. A. Cerino, and R. J. D. Jr, *Software Quality Control, Error, Analysis*. William Andrew, 1995.

[8] R. Binder, *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 2000.

[9] P. Aas, S. Dixit, T. Eden, L. Bruce, S. Moon, X. Wu, and S. O'Hara, *HTML 5.3: 4.12. Scripting*, Oct. 2018. [Online]. Available: https://www.w3.org/TR/2018/WD-html53-20181018/semantics-scripting.html#the-canvas-element

[10] *Percy — Visual testing as a service*. [Online]. Available: https://percy.io

[11] *ChromeDriver - WebDriver for Chrome*. [Online]. Available: https://chromedriver.chromium.org

[12] *JavaScript End to End Testing Framework — cypress.io*. [Online]. Available: https://www.cypress.io

[13] *RaiMan's SikuliX*. [Online]. Available: http://sikulix.com

[14] E. Borjesson and R. Feldt, "Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 350–359, Apr. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6200127/

[15] *SeleniumHQ Browser Automation*. [Online]. Available: https://www.selenium.dev

[16] *JUnit 5*. [Online]. Available: https://junit.org/junit5

[17] *Jenkins*. [Online]. Available: https://www.jenkins.io

[18] B. García, M. Gallego, F. Gortázar, and M. Munoz-Organero, "A Survey of the Selenium Ecosystem," *Electronics*, vol. 9, no. 7, p. 1067, Jul. 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/7/1067

[19] *OpenCV*. [Online]. Available: https://opencv.org

[20] *Tesseract OCR – opensource.google*. [Online]. Available: https://opensource.google/projects/tesseract

[21] *Instructor/Operator Station (IOS) Design Guide*. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/a192055.pdf

[22] *Teaching and Testing in Flight Simulation Training Devices (FSTD)*. [Online]. Available: https://www.easa.europa.eu/sites/default/files/dfu/206904_EASA_EHEST_HE_10.pdf

[23] *Instructor Station for FS2020, Prepar3D, FSX, FSW and X-Plane - FS-FlightControl*. [Online]. Available: https://www.fs-flightcontrol.com/en

[24] *Instructor Station — ALSIM*. [Online]. Available: https://www.alsim.com/simulators/instructor-station

[25] *Wetzel Technology GmbH - IOS*. [Online]. Available: http://www.wetzel-technology.com/Solutions/Operator_Station/operator_station.html

[26] D. U. Thibault, *Commented APP-6A - Military symbols for land based systems*, Sep. 2005, NATO Unclassified.

[27] D. J. Hemanth and U. Kose, *Artificial Intelligence and Applied Mathematics in Engineering Problems: Proceedings of the International Conference on Artificial Intelligence and Applied Mathematics in Engineering (ICAIAME 2019)*. Springer Nature, Jan. 2020.

[28] *OpenCV: Template Matching*. [Online]. Available: https://docs.opencv.org/4.4.0/de/da9/tutorial_template_matching.html

[29] R. Brunelli, *Template matching techniques in computer vision: theory and practice*. Chichester, U.K: Wiley, 2009.

[30] I. Pham, R. Jalovecky, and M. Polasek, "Using template matching for object recognition in infrared video sequences," *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pp. 8C5–1–8C5–9, Sep. 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7311477/

[31] ——, "Combining Template Matching and Background Subtraction Techniques to Detect Objects in Infrared Video Sequences," *Advances in Military Technology*, vol. 11, no. 2, Nov. 2016. [Online]. Available: https://apl.unob.cz/dam/20

[32] K. Subhalakhsmi and V. S. Soundharam, "Automatic License Plate Recognition System Based on Color Features and Vehicle tracking," *International Journal of Engineering Research*, vol. 3, no. 04, p. 4, 2015.

[33] *SikuliX - the basics — SikuliX 2.x+ documentation*. [Online]. Available: https://sikulix-2014.readthedocs.io/en/latest/basicinfo.html#sikulix-how-does-it-find-images-on-the-screen

[34] *tesseract-ocr/tessdata_best: Best (most accurate) trained LSTM models*. [Online]. Available: https://github.com/tesseract-ocr/tessdata_best

[35] C. Patel, A. Patel, and D. Patel, "Optical Character Recognition by Open source OCR Tool Tesseract: A Case Study," *International Journal of Computer Applications*, vol. 55, no. 10, pp. 50–56, Oct. 2012.

[36] *Improving the quality of the output — tessdoc*. [Online]. Available: https://tesseract-ocr.github.io/tessdoc/ImproveQuality

[37] M. Brisinello, R. Grbic, M. Pul, and T. Andelic, "Improving optical character recognition performance for low quality images," *2017 International Symposium ELMAR*, pp. 167–171, Sep. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8124460/

[38] *OpenCV: Adding (blending) two images using OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d5/dc4/tutorial_adding_images.html

[39] *OpenCV: More Morphology Transformations*. [Online]. Available: https://docs.opencv.org/3.4/d5/dc4/tutorial_adding_images.html