MASTER'S THESIS

Extending TESTAR's capabilities by integrating OCR for detecting textual presentation failures

Menting, T.

Award date:
2022

Extending TESTAR's capabilities by integrating OCR for detecting textual presentation failures

by

Tycho Menting

Presentation date:
25-05-2022

Student name:	Tycho Menting	
Student number:		
Title:	Extending TESTAR's capabilities by integrating OCR for detecting textual presentation failures	
Presentation date:	25-05-2022	
Program degree :	Open University of the Netherlands, Faculty of Science, Master's Programme in Software Engineering	
Course code:	IM9906	
Thesis committee:	prof. dr. Tanja Vos (chairwoman), dr. Pekka Aho (supervisor),	Open University Open University

Contents

Summary

Nowadays, most of the software applications which are being used by people contain a Graphical User Interface (GUI). A GUI simplifies the interaction with the application and makes it easier to present relevant information in a more structured way to the user. Testing software before releasing it is essential. By testing new versions, we can detect potential issues in advance. This gives the developers the opportunity to solve issues before releasing a new version and prevent a negative user experience. When a GUI contains a Presentation Failure (PF) the user experience is negatively affected. Depending on the severity of the PF, it is possible that the user misses important information. A type of PF is that text is not shown entirely. We refer to this type as Textual Presentation Failure (TPF). Testing GUI applications can be challenging and time-consuming because GUIs often change overtime. TESTAR is a test tool that can test GUI applications without having knowledge about the GUI. Optical Character Recognition (OCR) technologies have reached a mature state over the last decades. Nowadays, OCR is capable of identifying text located in images with a high accuracy. This thesis will integrate an OCR engine into TESTAR, to automatically validate that the text on GUI components is entirely visible for the user while testing the application. Based on the requirements, we have selected Tesseract as our OCR engine. After performing a proof of concept, we have successfully created an implementation of the OCR engine. The OCR engine returns the identified text from the captured screenshots, while TESTAR queries the expected text when testing the application. By comparing the expected text against the visualized text, our implementation can determine whether the text was visualized entirely or that we have detected a TPF. A controlled environment experiment has been executed to prove that the implementation is capable of detecting TPFs. Followed by validating the text visibility of three desktop applications and three websites with our new implementation. We have successfully enriched TESTAR with an OCR engine which we have used to validate that the expected text is visualized entirely.

Acknowledgement

I would like to express my gratitude to Tanja Vos for her assistance during my thesis and for her valuable comments, which she has given during the review session. I also want to express my gratitude to Pekka Aho for the feedback and guidance he has given while composing this thesis.

In addition, I also want to thank my parents, parents-in-law, sister, sister-in-law and in particular Fleur Riemersma for their unconditional support and giving me the possibility to invest the time needed to for executing the research and realising this thesis.

List of Figures

List of Algorithms

Chapter 1

Introduction

With the introduction of Graphical User Interface (GUI) applications in computer environments, the possibilities to visualize content and the way how a computer program could inform the user, have increased tremendously. Complex information structures can be presented in a more convenient way, since there is more space available compared to a single command line. Also, the possibilities to interact with the computer have increased with this introduction. Concepts such as buttons, checkboxes and radio buttons extended the application input capabilities.

One of the problems that a GUI application could encounter is a layout anomaly, also known as a Presentation Failure (PF) [1]. PFs can result in different visual failures. The following list is not exhaustive. It provides an overview of common PFs.

- Cropped text.
- Missing GUI widget.
- Overlapping GUI widgets.
- Overflowing GUI widgets. In this case, the widget is partially off screen.
- Misalignment of GUI widgets.
- GUI widgets scale maladaptation. In this case, the widget does not follow the minimum and maximum dimensions specified by the GUI guidelines.
- Text scale maladaptation.

Introduced PFs can have different causes. A common cause is that the application runs on different screen configurations. If the developer did not properly implement the GUI by ignoring the scaling capabilities of the used framework, it could result in having PFs within the application. Another potential cause could be by having a mismatch in the required dimensions to visualize the text and the actual size for a particular text element in combination with a different presentation language. This is known as an Internationalization Presentation Failure (IPF). Translations of presented text elements do not necessarily need to have the same length as the original text. In case that the translation is longer than the original text and if the GUI element that is responsible for visualizing the text is not adjusting to the required dimensions of the translated text, an IPF can be expected. Translations of websites and applications do not necessarily have to be created during the development phase. It is very common to place

the translations into resource files that can be loaded dynamically. This makes it possible the IPFs can be introduced after an application has been released. Websites can even be translated directly by using an external service, such as Google[1].

The user experience is negatively affected when a website or application contains PFs. Especially when the PF involves text which is not presented entirely. The impact of such a PF can range from a minor visualization error, like a truncated last letter, to a text from which the intention is not understandable. For all PFs which contain text, we introduce a new category called Textual Presentation Failure (TPF). TPFs are not related to the cause. This means that both an IPF and an overlapping widget can result in introducing a TPF. Figure 1.1 shows an overview of the PFs are linked to TPFs.

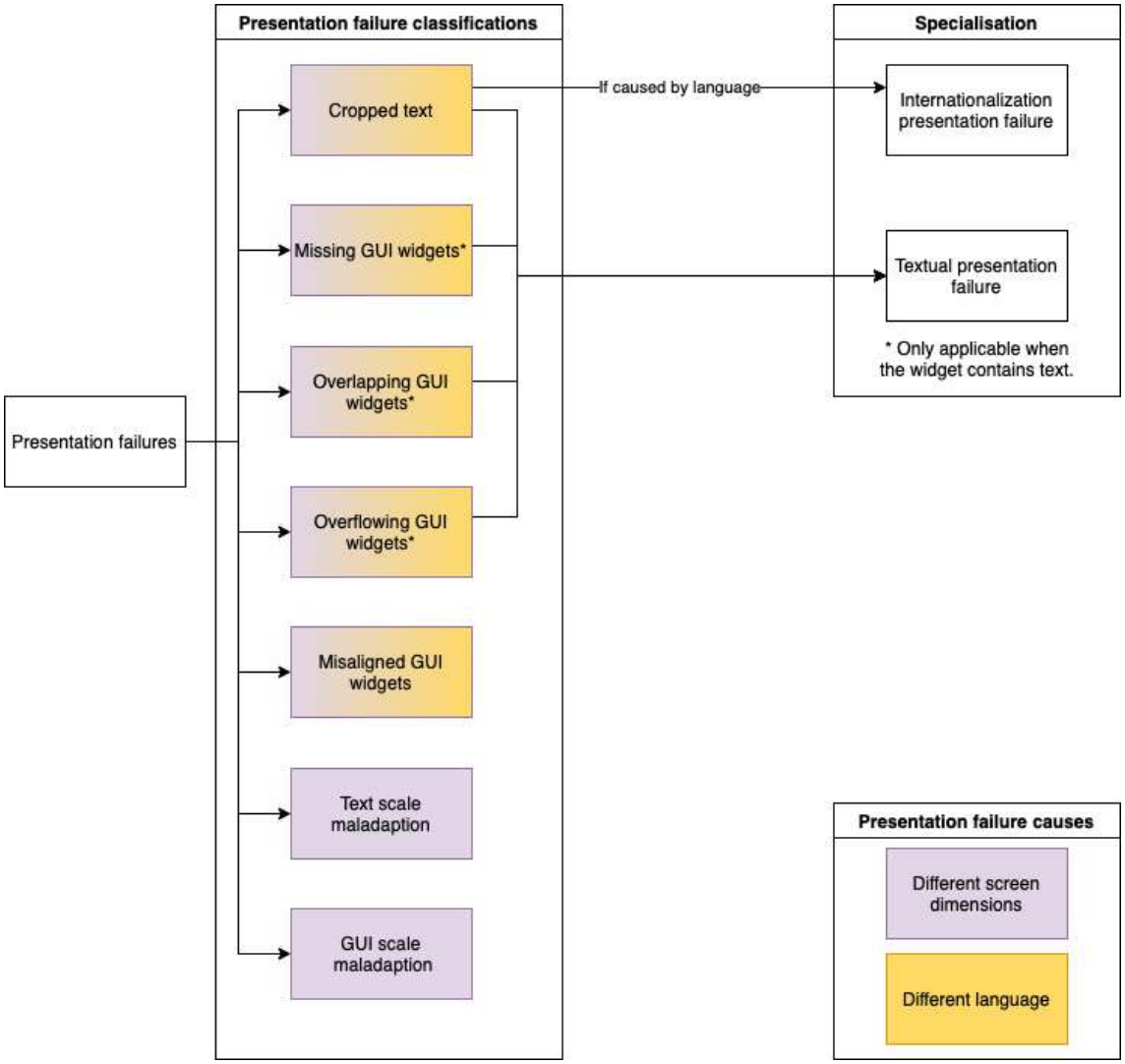


Figure 1.1: Overview of PFs.

Testing applications has become more challenging now that applications can contain a GUI. Traditional unit tests will only cover the logic of an application, but now that applications can visualize information, we need to make sure that the content we want to show is presented correctly. One way of accomplishing this is by comparing a correct screenshot with the actual composed view of the application [2]. This is however subject to change, every change in the GUI results in updating the reference images. When the comparison test is also responsible

[1] `https://cloud.google.com/translate/`

for validating that the GUI is presented correctly among the different supported languages, updating the reference images is labor intensive.

To prevent PFs from happening, the first line of defense is guidelines [1]. The problem with guidelines is that they only make the developer aware of the problem, but they can not detect them. Awareness is created by providing a set of best practices. Apple offers a development tool[2], but this requires a manual inspection for each view and only applies to Apple specific applications. For website development there are two related techniques that can detect PFs. The first solution uses reference images to detect PFs [3]. The downside of this approach is that reference images need to be kept up-to-date when the GUI changes. The second approach is based on a reference version of the same website [1]. This solution requires the test to be updated when the GUI changes.

TESTAR is a software testing tool that is capable of executing actions on a System Under Test (SUT) and validating whether the outcome contains unexpected results. In this work we will use TESTAR for testing the SUT. When we refer to concepts or code of TESTAR, we will use the following style; "*Hello world*". TESTAR can test without having knowledge about the structure of the application. Based on the type of application, TESTAR can obtain the actionable elements via an interface called "*protocol*". For a desktop application the protocol is hooked up to the Windows Accessibility Application Programming Interface (API)[3] and for a website the protocol makes use of the Selenium webdriver[4]. Validating the outcome is accomplished by comparing whether the outcome matches one of the predefined oracles. During a test run, TESTAR automatically creates screenshots of the SUT before the execution of an action and an isolated screenshot containing only the executed action. TESTAR is also capable, based on the type of SUT, to obtain the expected text from the protocol. These two functionalities contribute in detecting TPFs in an automated way. The automated capture of screenshots eliminates the need to manually create new screenshots when the GUI has changed and shows the text that the application has visualized. Depending on the SUT, TESTAR uses the protocol that is capable of extracting the expected text.

To automate the validation process, we can extract the text from the captured screenshots using Optical Character Recognition (OCR) and compare that with the expected text. The goal of this research is to investigate:

> *How can we implement OCR into TESTAR for extracting the text from the screenshots and create an automatic validation algorithm for detecting TPFs.*

With this research we aim to reduce the required test effort for detecting incorrectly visualized text elements for both websites and desktop applications. By integrating OCR we can close the loop and try to automatically validate whether the text is presented correctly. TESTAR gives us the advantage that we do not need to update the tests once the GUI changes. When we succeed in developing the automated validation mechanism, detecting TPFs in GUI applications can be automated. This makes it possible for the developers to focus even more on the actual development of the application.

In chapter 2, we introduce a proof of concept and select an OCR engine for our implementation. The outcome of the proof of concept and an overview of the research questions will be covered in chapter 3. For the technical design and the implementation of the OCR engine and the

[2] `https://help.apple.com/xcode/mac/current/#/dev499a9529e`
[3] `https://docs.microsoft.com/en-us/windows/win32/winauto/windows-automation-api-portal`
[4] `https://github.com/SeleniumHQ/seleniumhq.github.io`

validation mechanism, we refer to chapter 4. In chapter 5, we perform a exploratory experiment to validate the implementation and discuss the results. Chapter 6 covers the validation of our implementation where we will test six applications and discuss the threats to validity. We dedicate the final chapter 7 to the conclusion and we will discuss the future work.

1.1 Optical Character Recognition (OCR)

OCR makes it possible to extract characters from handwritten or printed documents. With the extracted characters it becomes possible for a computer to process the information further. We could use it, for example, to create an index of all the discovered words, which can be used in a search function to lookup a particular word. OCR engines are tasked with a challenging problem. It can be difficult to distinguish between the characters, because they all look similar, for example 'l', '1', 'I' and '|'. This is especially troublesome for handwritten documents, since these often contain inconsistencies when it comes to letter size and spacing. In the beginning, OCR engines had problems with detecting the text correctly, the error margin was high and the detection rate was low. Nowadays, the performance has increased significantly and the error margin has dropped [4]. While OCR engines used to be local systems, it is now also possible to use an online OCR engine for obtaining characters from an image or document. Powerful servers run the OCR engine and will report the outcome, for example via the web browser. The input flows through three stages while it is being processed by an OCR engine. First stage is the pre-processing stage. In this stage the input is prepared for examination by the engine. Operations like normalizing the scale and aspect ratio of the input and rotating the input so that is properly aligned are part of the first stage. In the second stage, the character recognition takes place. Here the individual characters are being detected based on their outline. The last stage is the post-processing stage. Detected characters are grouped and examined in this stage to compose words. Some examples of areas where OCR is being applied are:

- Automatic License Plate Recognition (ANPR) [5].
- Accessibility tools for blind and visually disabled people.[6]
- Information extraction on official documents, like passport inspection performed by border patrol.[7]

1.2 Testing

1.2.1 Automated testing

From the first line of code until the end-of-life phase of a project, validation of the code is important. We can accomplish this by testing that the code does what it is supposed to do. Over time, it can become too complex to validate the codebase manually. It could be the case that testing the codebase manually consumes a lot of time, but also manually testing the codebase is error prone. When manually executing the test multiple times, the chance to execute

[5]https://www.beltech.nl/portfolio/automatic-number-plate-recognition-anpr/

[6]https://www.afb.org/blindness-and-low-vision/using-technology/assistive-technology-videos/scanning-and-specialized-ocr

[7]https://nanonets.com/case-study/ocr-passport

the test in a slightly different way than the previous run increases [5]. Repeatedly performing the same actions affects the testers concentration.

Automating the testing activities comes with several benefits. First, it saves a lot of time, executing the test does not require the presence of the tester, this saves a lot of manual labor. Only when the environment in which the software is running has changed, or when changes to the code have been made affecting the behavior of the program, a test should be updated. But as long as the code and its environment stay consistent, we just need to write the test once. We can repeat the execution of the test as many times as we need on different versions of the software [6]. Second, we execute the test in a consistent way. Each execution shall follow the exact same steps as in the previous runs, except when we explicitly use randomness.

We can apply automated testing in different areas of the software development. We can use it on the lowest level with a narrow scope, like unit testing, automating testing of a small piece of code in isolation. But we can also use it on the highest level, like for example on acceptance testing where we test the final product.

1.2.2 Introduction to GUI testing

In the early days of the computer, the user could only interact with the application via the command line. In 1973 Douglas Engelbart [7] was working for PARC and created the Alto computer. This was the first computer that had a screen on which the pixels could be controlled individual. With this new screen it became possible to create GUI applications. One of the first GUI applications was visualizing the file architecture on the Alto computer. With introducing this new groundbreaking technique more and more computer applications have a GUI nowadays. Figure 1.2 shows the Alto file manager.

Figure 1.2: Screenshot of the Alto file manager.

With this new way of interacting with an application, another level of complexity has been added in order to create a stable application. The GUI adds a lot of new states, for example, when additional input fields or input options are used in a normal operation flow when they should not be used. With all these new interaction options, the states of the application have increased exponentially. This makes it harder to test whether an application is still working as expected.

Computer programs keep evolving during their lifecycle because of new requirements, bugs that need to be solved or changing environments [8]. All these reasons involve modifications on the source code. When these changes are not being applied carefully, introducing new bugs is almost unavoidable. Maintaining computer programs do require additional test effort in order to make sure that regression has not been introduced and the the computer program behaves as expected [9].

Reducing the test effort can be achieved by implementing test automation. Instead of manually clicking through the GUI of the application and verifying that application works properly, a lot of time can be saved by executing the tests automatically. With the help of test automation, the user needs to create a test once and can execute the test multiple times on each new version of the application automatically. There are multiple techniques that can be used to automate GUI testing.

A technique that is being used by a large amount of GUI testing tools is Capture and Replay (C/R) [10]. The test is being composed by recording the user's input while running the application. Testing for regressions on a newer version of the application can be done by replaying the test. One disadvantage of the C/R technique is that once the GUI has changed, the test needs to be repaired by recording the scenario again with the new GUI version of the application.

Other tools are using a technique called Visual Testing [11]. This is a combination of scripting and image processing. The test case is composed out of screenshots of the different actions and the script holds the required action information. For example, a screenshot of a button with a desired user interaction *click*. When the test is executed, the tool uses image processing to search for a match with the screenshot of the button. Once found, the action shall be applied. The downside of this technique is that it is not accurate in identifying the right elements which results in false positives, and it is slow because of the image processing.

Another technique is Model-Based Testing [10]. By creating formal models of the application and determining the different input and output scenarios, test cases can be generated automatically with this technique.

A problem with GUI applications is that when the GUI has changed, depending on the used technique, the tests also need to be updated. This leads to additional work that increases the test effort. There are tools that can handle GUI changes in a proper manner, for example TESTAR.

1.2.3 Introduction to TESTAR

The development of TESTAR tool has started with the funding of the European Union FITTEST project [12]. Automatically testing for crashes is one of the capabilities of TESTAR. Test cases that are being executed by TESTAR are not immediately affected when the GUI has changed, because TESTAR is scanning the GUI state while running the test. Depending on the selected protocol, the tool is scanning the GUI for components via the Accessibility API

or the webdriver, in this way the tool can execute test cases for native applications as well as testing web pages.

When TESTAR is using the Accessibility API from the operating system, information is being collected about the GUI components that are currently displayed on the screen. This could be for example: size, text, position, parent component. With this information, TESTAR is composing a "*widget tree*" containing the GUI component information of the application for that particular state. All these states are stored inside a state model, and based on this model, follow-up actions are being generated by the tool. When the test case is executed, an action is selected based on the action selection strategy and executed. For example, provide input or click on a button. Based on the triggered action, the result of the execution can be validated with an oracle for its correctness [13]. And finally TESTAR can decide to proceed with the test if there are more actions that can be executed or to stop the test.

1.3 Textual presentation failure

Since there is not a common language that everybody around the world understands, chances are higher that a user understands the intention of an application when it comes with built-in support of the native language of the user. Before the introduction of the Internet, sharing an application with someone else across the world was difficult. Therefore, most of the applications were presented in the native language of the user, so they could be understood by the user. But after the introduction of the Internet, sharing an application became relatively simple and people started to use applications from all around the world. So, with this new possibility, the question for multi-language applications has arisen. Also, with the introduction of smartphones, mobile applications need to support a large variety of different screen sizes. When an application with a fixed layout size for screen 1 is shown on a smaller screen 2, the content may not be presented entirely. As explained in section 1, changing the presentation language of the application or website could introduce a TPF. Figure 1.3 shows a screenshot of a TPF caused by switching the presentation language.[8].

Figure 1.3: Example of a TPF.

Multi language applications and the need to run an application on different hardware devices make it very difficult to test the GUI for the presence of TPFs. Automating the validation of the expected content with the content that is actually presented on the screen can prevent releasing a new version of the application which contains TPFs.

[8] `https://github.com/nextcloud/server/pull/7167`

1.4 Related work

Applications can run in various environments these days. Android is one of the major operating systems for mobile devices. The different screen sizes and densities are one of the challenges when it comes to visualizing the application correctly. Because of a large variety of mobile devices, it becomes very difficult for a developer to validate that the application is shown correctly on each device. We found three tools that can test Android applications for multiple PFs that we will discuss next. Figure 1.1 shows an overview of the multiple PFs.

The `PATS` [14] tool has a validation process that consists of detecting missing widgets and cropped text. This is accomplished by using a reference screen size and density. For the text validation, `PATS` uses OCR to extract the visualized text while the expected text is obtained via the XML layout data which is extracted from the SUT via a GUI ripping module. This approach is similar compared to our research. The main difference is that for `PATS` only supports Android applications and no desktop applications or websites. In addition, the three android applications which are used to validate the implementation have a very simple layout. Only one application has been downloaded from the Google Play Store, while the other two applications are custom developments without a large install base. In our research, we will use more complex applications.

A similar tool is the `LAD tool` [15]. The major difference compared with `PATS` is that this tool has extended functionality and can detect four more PFs. This includes text scale issues, overlapping widgets, misaligned widgets and widget overflow. For detecting TPFs, the tool uses the same approach as `PATS`. The `LAD tool` does also only support android applications, where our research will focus on desktop applications and websites.

Another framework for detecting text visualization failures in Android applications is `Textout` [16]. This framework uses a convolutional neural network (CNN) to detect incorrectly visualized text elements. The CNN has been trained to recognize text layout failures when it analyzes the screenshot snippet if that contains the text element. The major difference with this approach is that it can only identify if text is not visualized correctly. It does not know what the expected text should be, so comparing the visualized text against the expected text is not possible.

When it comes to testing websites for PFs we found existing tools described next.

`ReDeCheck` [17] is a tool which can test responsive websites for their consistency and the presence of five different types of PFs. This is accomplished by creating a model from the HTML after querying the Document Object Model (DOM) of the website. However, none of the PF types covers the validation of the visualized text. In our research, we focus only on validating whether the expected text is visualized entirely.

Another tool called `GWALI` [1] can also validate websites. `GWALI` creates a model containing the relative position and visual relation between HTML tags and text elements. It uses a reference version which has already been manually inspected and considered being correct. The extracted model of the reference version is then compared to a model of the same website with a different presentation language. By comparing the two models, the tool is capable of identifying IPFs. While `GWALI` is capable of detecting IPFs, it is not capable of validating whether the expected text is visualized correctly.

A prototype tool [3] has been created to identify PFs based on image comparison. This tool creates two screenshots, one from a reference version of the website and another one screenshot is taken from the version under test. Based on a pixel comparison, the visual difference between

the two screenshots is being calculated. When a visual difference has been identified, the location of this difference is used to locate the corresponding HTML tag which is responsible for the visualization. The downside of this approach is that the reference images need to be updated when the layout of the website has changed. Because TESTAR composes a widget tree dynamically, it is not affected by layout changes. Also, this prototype tool only validates the website by using a reference image which needs to be inspected manually and does not automatically validate if the expected text is shown correctly.

Up next we describe the tools found for testing whether PFs are present in desktop applications.

A prototype tool [18] validates whether graphical objects used in windows applications are presented properly. This is accomplished by intercepting the API draw commands when the screen is composed. With the intercepted draw commands, the object can be regenerated and used as a reference to compare the graphical object across different versions. This prototype tools only validates the visualization of graphical objects while our research will focus on whether the text is visualized correctly.

Ramler has performed multiple researches which are related to PF detection for desktop applications. For these researches, the SUT was a private software suite from `OMICRON electronics` [9]. Dedicated scripts were created to navigate through the SUT. While navigating through the SUT, screenshots are captured and relevant widget information is stored by querying the API of the GUI test automation tool. One research [19] is dedicated to detection of multiple PF types across different pixel densities. However, validating whether the expected text is shown correctly is not included. The PF detection is accomplished by scaling the captured screenshot so that pixel comparison becomes possible. Together with a comparison of the relevant widget information, PFs can be detected. In another research [20], the SUT is tested for both PFs and IPFs. This is done by checking the size and position of the bounding box of the widget. Also, in this research, the OCR engine is not used to inspect the visualized text.

1.5 Research methodology

In this thesis, we will focus on the problem described in the previous section. We will implement a solution for automatically testing whether the presented text of GUI applications is shown correctly on the screen. We will do this by implementing a new visual validation module into TESTAR. This new module will make use of an OCR engine to help with the validation process. OCR makes it possible to detect characters in images. The new module shall take screenshots of the application under test while navigating through the GUI. With the help of the accessibility API or the webdriver, TESTAR knows the text that should be presented and by comparing the results of the OCR, we can validate if the text is presented correctly.

We will start by selecting an OCR engine based on the requirements we have defined. The chosen OCR engine will then be integrated into TESTAR. This integration is mainly to be able to perform a proof of concept. Therefore, the OCR integration is subjected to change. The proof of concept contributes to our design of the new visual validation module. Problems encountered during the proof of concept will be taken into account during the design phase. The outcome of the proof of concept will also be used to refine our research questions. Up next, we will, based on the created design, implement the new visual validation module into TESTAR. We will elaborate for each component within the new design on how it contributes to our

[9] `https://www.omicronenergy.com`

implementation. The new module will be first tested by performing an exploratory experiment. For this experiment, we will manipulate an application in such a way that it contains TPFs. After refining and testing the visual validation module, we will validate the implementation with different applications and websites. We will create a metrics which will be used during the validation process. The outcome of each application will be discussed, and the defined metrics will be applied. The implementation of the visual validation module, the configurations used for testing the different applications and websites and the outcome of the validation, will be stored in a publicly accessible location.

Chapter 2

Selecting an OCR engine

In this chapter, we will introduce the requirements for the OCR engine. Based on the requirements, we will create a scoring card. The scoring card will then be used to select the OCR engine which we will implement into TESTAR.

2.1 OCR requirements

Before we could start with the integration of an OCR engine, we needed to make a decision on which OCR engine we would be using for this research. There are a lot of OCR engines on the market. In order to select an engine, we defined a set of requirements. In addition we also defined a scoring table, we distributed weight factors on the topics that were important for our implementation. And after rating the OCR engines, we selected the OCR engine with the highest score. We took the following requirements into account during the selection process.

License
TESTAR is an open source project which is free to use. Development and maintenance is being done by the university of Valencia and the Open University in the Netherlands. Neither universities are planning to sell TESTAR for commercial purpose nor do they plan to make a profit out of it. It is therefor mandatory that the OCR engine is free from any form of license cost, so that the development cost stay as low as possible.

Availability
Regarding the availability, there are several options; The first option is that the OCR engine runs completely offline (without Internet connection). The second option is on the other side of the spectrum, the OCR is only available as an online service. Or the last option, a hybrid solution in which we use offline as a fallback scenario when the connection with the online service is not available. TESTAR does not currently depend on any online service whatsoever. Therefor it is desirable to keep it as is, this requires that the OCR engine must be available offline.

Implementation
This research is focusing on the question whether or not having OCR can be beneficial for TESTAR. To prevent taking a side road, the integration with TESTAR must not become a bottleneck. Therefor it is required that we can easily implement the OCR engine in the current codebase of TESTAR. We can achieve this by using an OCR engine which has been written in

the same language, or comes with a wrapper which makes it possible to access the OCR engine in the native language of TESTAR. An alternative could be that we append the OCR engine as a standalone implementation to the installation package.

Accuracy
To contribute to a thorough test run of the SUT, it is required that the OCR engine is accurate. Having an OCR engine with a low accuracy will inevitable result in failures. By feeding the different OCR engines the same test set, we can determine the accuracy relative to each other.

Language support
The number of supported languages that the OCR engine supports is also an important point of distinction. For this thesis we only require that the OCR engine can parse English and one other language, but the more languages the OCR engine supports the better. In addition, it is even better if the OCR engine contains the feature which makes it possible to extend the supported languages. In this way, the users can add the language they need if it is not included by default. This could also potentially allow the user to improve a specific language by providing an improved version of the language recognizing dataset.

Support
A minor topic is support. In the scenario where we can not integrate the OCR engine out of the box, or when we need to fine tune the configuration of the OCR engine, we need to be able to consult a source of help. We have defined three indicators that contribute to the support topic. The first one is the current phase in the software lifecycle of the OCR engine. An OCR engine that is still actively maintained scores better compared to an engine that has stopped the development. Secondly, we rate whether the engine has a large community. Having such a community increases the change that they have solved a problem before. The last indicator is documentation, having documentation about how to use the OCR engine is beneficial for working with the OCR engine. A rich set of examples is also valuable to have.

2.2 OCR scorecard

Based on the requirements, we selected an OCR engine that satisfied the criteria. We graded each OCR engine per category and selected the OCR engine with the highest score for this research. Table 2.1 shows an overview of the grades per category. In the next section, we will explain how we decided upon the different grades.

The license and availability are the two most important requirements, followed by the ease of implementation effort. We do not want to introduce new constraints for using TESTAR. So introducing a license fee for the OCR functionality is not desired. Therefore, the free versions are scoring higher compared to the paid options. Also, we do not want to introduce the requirement of having an active internet connection while running TESTAR. However, it can be beneficial to have an OCR engine that can switch to an online engine when needed. So the hybrid solution has the highest score, followed by the offline only option. OCR engines that require an active internet connection do not get any points at all on this topic. An OCR engine that provides a Java based SDK scores the highest. This is because TESTAR is also written in Java. The required effort for integrating the OCR into the codebase of TESTAR will be reduced to a minimal since we do not need to write a wrapper in order to interact with the engine. A minor requirement is language support. An OCR engine that has language support is better capable of detecting the right words instead of detecting individual characters. Having language support provides context to the OCR engine. The discovered characters can be grouped and matched to

Category	Option	Grade
License	Paid	0
	Free	15
Availability	Online	0
	Offline	10
	Hybrid	15
SDK language	Java	10
	Other	5
	None	0
Language support	0	0
	1 - 15	5
	15+	10
Accuracy	95+	5
	80 - 95	2
	$<$ 80	0
Support:		
Development	End of life	0
	Maintenance	2
	Active development	5
Community	Small	0
	Large	2
Documentation	Poor	0
	Good	2

Table 2.1: OCR Scorecard

known words of the selected language. The more languages are supported by the OCR engine, the more beneficial it is for TESTAR. Having a wider range of supported languages, the broader TESTAR can validate applications that use a different language. We have selected the following categories for the first grading round, license, availability, SDK language and language support. The highest score is 50 points. These categories are the most important topics for selecting an OCR engine. For OCR engines that differ 5 or less points, we will investigate how the engines grade on the topics of accuracy and support.

2.3 OCR engine grading

Engine	License	Availability	Implementation	Language support	Score
Adobe Acrobat[21]	Paid (0)	Hybrid (15)	No SDK (0)	13 (5)	20
ABBYY Cloud OCR SDK[22]	Paid (0)	Online (0)	REST API (5)	200+ (10)	15
ABBYY FineReader Engine[23]	Paid (0)	Offline (10)	Java (10)	210 (10)	30
Amazon Textract[24]	Paid (0)	Online (0)	Java (10)	1[1](5)	15
Asprise OCR[25]	Paid (0)	Offline (10)	Java (10)	23 (10)	30
Calamari[26]	Free (15)	Offline (10)	Python (5)	6 (5)	35
CuneiForm[27]	Free (15)	Offline (10)	C++ (5)	24 (10)	40
Dynamsoft[28]	Paid (0)	Offline (10)	C# (5)	40+[2](10)	25
GOCR[29]	Free (15)	Offline (10)	C (5)	0[3](0)	30
Google Cloud[30]	Paid (0)	Online (0)	Java (10)	60 (10)	20
Kraken[31]	Free (15)	Offline (10)	Python (5)	10 (5)	35
Azure's Computer Vision API[32]	Paid (0)	Online (0)	REST API (5)	7 (5)	10
OCRvision[33]	Paid (0)	Offline (10)	No SDK (0)	99 (10)	20
OmniPage[34]	Paid (0)	Offline (10)	Java (10)	125 (10)	30
OpenText[35]	Paid (0)	Offline (10)	C++, C# (5)	6+ (5)	20
Ocrad[36]	Free (15)	Offline (10)	C++ (5)	0[3](0)	30
Ocropy[37]	Free (15)	Offline (10)	Python (5)	13 (5)	35
Ocular[38]	Free (15)	Offline (10)	Java (10)	0[4](0)	35
Readiris[39]	Paid (0)	Offline (10)	C++, C# (5)	138 (10)	25
Rossum[40]	Paid (0)	Online (0)	REST API (5)	5 (5)	10
SimpleOCR[41]	Paid (0)	Offline (10)	C++, C# (5)	4 (5)	20
Transym[42]	Paid (0)	Offline (10)	C, C# (5)	11 (5)	20
Microsoft UWP OCR[43]	Free (15)	Offline (10)	C# (5)	1[5](5)	35
Tegaki[44]	Free (15)	Offline (10)	Python (5)	12 (5)	35
Tesseract[45]	Free (15)	Offline (10)	Java (10)	100+ (10)	45

Table 2.2: OCR engines

[1] Currently only English.
[2] Additional languages come from tesseract.
[3] Detects character only.
[4] Need to train self.
[5] Based on installed language pack.

Table 2.2 shows scoring overview of each OCR engine. Based on the ranking, we looked at CuneiForm and Tesseract since they were the most interesting engines. When we took the support requirement into account, we noticed that Tesseract is actively being maintained while the last changes to the codebase of CuneiForm date from 2011. Also, the community of Tesseract is bigger and looking at the codebase the documentation scores better compared to CuneiForm. The gap between Tesseract and the other candidates was too big in order to make a difference in selecting the OCR engine. Based on the test results[1], we decided to not perform any addi-

[1] `https://github.com/tesseract-ocr/docs/blob/master/das_tutorial2016/7Building%20a%20Multi-Lingual%20OCR%20Engine.pdf`

tional accuracy test. In addition, Tesseract engine supports configuration and can therefor be optimised when needed. Based on the grading, we concluded that Tesseract is the best match for our research and will therefor be used as OCR engine for our research.

Chapter 3

Refining the research questions

This chapter is devoted to the proof of concept and the outcome of it. Based on the results we will refine the research questions.

To explore the area of interest for the main research question, we started with a proof of concept. The goal was to implement an OCR engine into TESTAR. This gave us insights into the architecture of TESTAR and an overview of the components with which we needed to interact with to integrate the OCR engine. Also, it allowed us to identify potential problems in an early stage of the project. The sooner we identify the problems the easier it is to create a proper solution that is inline with the final implementation of the OCR engine. We identified the following topics that we will investigate in more detail while working on the proof of concept.

- Integration of an OCR engine.
- Creating a screenshot of the SUT.
- Obtaining the text that is expected to be presented from TESTAR.
- Comparison of the OCR results with the expected text from TESTAR.

3.1 Problems encountered during the proof of concept

At the start of the proof of concept, TESTAR could already take screenshots from the SUT. To be even more precise, we can configure TESTAR to take an overall screenshot of the SUT before or after an action is being performed and it can also take a screenshot of the action itself in isolation. After performing a test run, we discovered that the screenshots were not well aligned with the area of interest, see figure 3.1. To solve this, we started an investigation to find the root cause of this problem. We noticed that the test run was executed on a monitor that used display scaling to show the content. Display scaling increases the readability on high resolution displays, creating larger virtual pixels by combining physical pixels, also known as the effective pixel. The screenshot functionality of TESTAR did not take the display scale into account while creating the screenshots. This resulted in the use of the raw pixel dimensions instead of the effective pixel dimensions. Looking up the display scale for the monitor that presents the SUT from the operating system solved the problem, see figure 3.2. While fixing this problem we gained knowledge about how TESTAR stores the widget information, like text

and ancestors and how the accessibility API is being used to get the actual position for specific widgets.

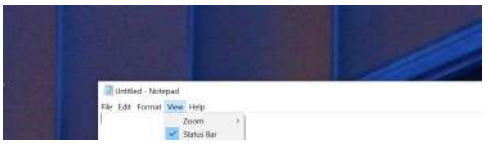


Figure 3.1: Misaligned screenshot.

Figure 3.2: Aligned screenshot.

3.2 Findings of the proof of concept

Based on the outcome of the OCR selection in 2.3, we have integrated Tesseract as OCR engine. By creating a dedicated abstract interface for all the OCR functionality, we make it possible, at a later stage, to perform comparison experiments with other OCR implementations. For the proof of concept, we have chosen the solution that would require minimal effort to achieve a result. This was the solution where we reuse the stored screenshots on disk and feed them to the OCR engine. In the final implementation, we should process the captured screenshots while they are loaded in memory. This eliminates the time we need to wait until the screenshots are saved to disk. The first result of the OCR engine is promising. We have tested two SUT's, the `Notepad` desktop application, by using the desktop generic protocol. The other SUT was the `www.ou.nl` website for which we used the default webdriver protocol. We noticed that the OCR results for the default webdriver protocol returned way more detected words than that the webpage `www.ou.nl` was showing. Figure 3.4 contains a part of the OCR results which includes text elements which are not visible on the webpage. Figure 3.3 shows a selection of the OCR results and figure 3.5 shows the actual website that was used during the test.

```
Word: 'Deze';   conf: 96.47; BoundingBox: 389,1448,420,1458;
Word: 'website';    conf: 93.03; BoundingBox: 425,1448,475,1458;
Word: 'gebruikt';   conf: 90.69; BoundingBox: 480,1447,535,1462;
Word: 'cookies';    conf: 90.02; BoundingBox: 540,1447,589,1458;
Word: '(en';    conf: 93.23; BoundingBox: 594,1449,613,1459;
Word: 'daarmee';    conf: 82.37; BoundingBox: 619,1447,678,1458;
Word: 'vergelijkbare';      conf: 84.53; BoundingBox: 683,1447,768,1462;
Word: 'technieken)';    conf: 92.87; BoundingBox: 773,1447,850,1460;
Word: 'om';     conf: 96.86; BoundingBox: 855,1450,875,1458;
Word: 'het';    conf: 93.26; BoundingBox: 881,1448,902,1458;
Word: 'bezoek';     conf: 93.07; BoundingBox: 907,1447,953,1458;
Word: 'voor';   conf: 93.31; BoundingBox: 958,1450,986,1458;
Word: 'u';      conf: 93.16; BoundingBox: 993,1450,999,1458;
Word: 'nog';    conf: 93.26; BoundingBox: 1006,1450,1029,1462;
Word: 'makkelijker';    conf: 91.63; BoundingBox: 1035,1447,1108,1462;
Word: 'en';     conf: 93.31; BoundingBox: 1115,1450,1130,1458;
Word: 'persoonlijker';      conf: 86.73; BoundingBox: 1136,1447,1221,1462;
Word: 'te';     conf: 96.96; BoundingBox: 1228,1449,1240,1458;
Word: 'maken.';     conf: 95.41; BoundingBox: 1245,1447,1289,1458;
Word: 'Met';    conf: 95.41; BoundingBox: 1299,1448,1324,1458;
Word: 'deze';   conf: 95.88; BoundingBox: 1328,1447,1359,1458;
Word: 'cookies';    conf: 96.57; BoundingBox: 1364,1447,1413,1458;
Word: 'kunnen';     conf: 96.72; BoundingBox: 1418,1447,1467,1458;
Word: 'wij';    conf: 84.73; BoundingBox: 1472,1450,1489,1462;
Word: 'en';     conf: 84.73; BoundingBox: 1495,1450,1510,1458;
Word: 'derde';      conf: 87.15; BoundingBox: 1516,1447,1554,1458;
```

Figure 3.3: OCR result contains single words only.

```
OCR has found:
Word: 'elma';   conf: 11.22; BoundingBox: 2063,19,2221,40;
Word: '-';      conf: 17.02; BoundingBox: 2862,10,2873,16;
Word: 'iu';     conf: 0.00; BoundingBox: 519,74,603,186;
Word: 'Open';   conf: 96.38; BoundingBox: 605,103,669,150;
Word: 'Universiteit';   conf: 96.38; BoundingBox: 679,107,834,146;
Word: 'Onderwijs';      conf: 46.00; BoundingBox: 1453,120,1564,144;
Word: 'v';      conf: 37.36; BoundingBox: 1595,126,1608,134;
Word: 'Onderzoek';      conf: 89.42; BoundingBox: 1655,120,1776,138;
Word: 'v';      conf: 89.42; BoundingBox: 1807,126,1820,134;
Word: 'Overons';    conf: 90.88; BoundingBox: 1867,121,1966,138;
Word: 'v¥';     conf: 32.43; BoundingBox: 1996,126,2010,134;
Word: 'Contact';    conf: 90.84; BoundingBox: 2057,121,2141,138;
Word: 'v';      conf: 78.61; BoundingBox: 2171,126,2184,134;
Word: 'Q';      conf: 93.12; BoundingBox: 2245,119,2267,141;
Word: ';';      conf: 12.98; BoundingBox: 590,206,595,208;
Word: '4';      conf: 52.92; BoundingBox: 819,206,823,208;
Word: '-';      conf: 56.07; BoundingBox: 930,206,938,210;
Word: '¥';      conf: 0.00; BoundingBox: 1086,206,1099,213;
Word: 'ar';     conf: 37.72; BoundingBox: 1581,211,1594,220;
Word: 'a';      conf: 15.51; BoundingBox: 1609,209,1627,221;
Word: '&';      conf: 0.00; BoundingBox: 2092,219,2100,228;
Word: 'be';     conf: 20.78; BoundingBox: 2202,218,2210,228;
```

Figure 3.4: False positives webdriver protocol.

A quick investigation showed that the OCR engine detected characters that did not make valid words, as we can see in figure 3.4. Also, the OCR results for the "*desktop generic protocol*" contained false positives. For example, the OCR engine identified the maximize window button in the upper right corner as an 'O' character, while this is actually the shape of a rectangle, as

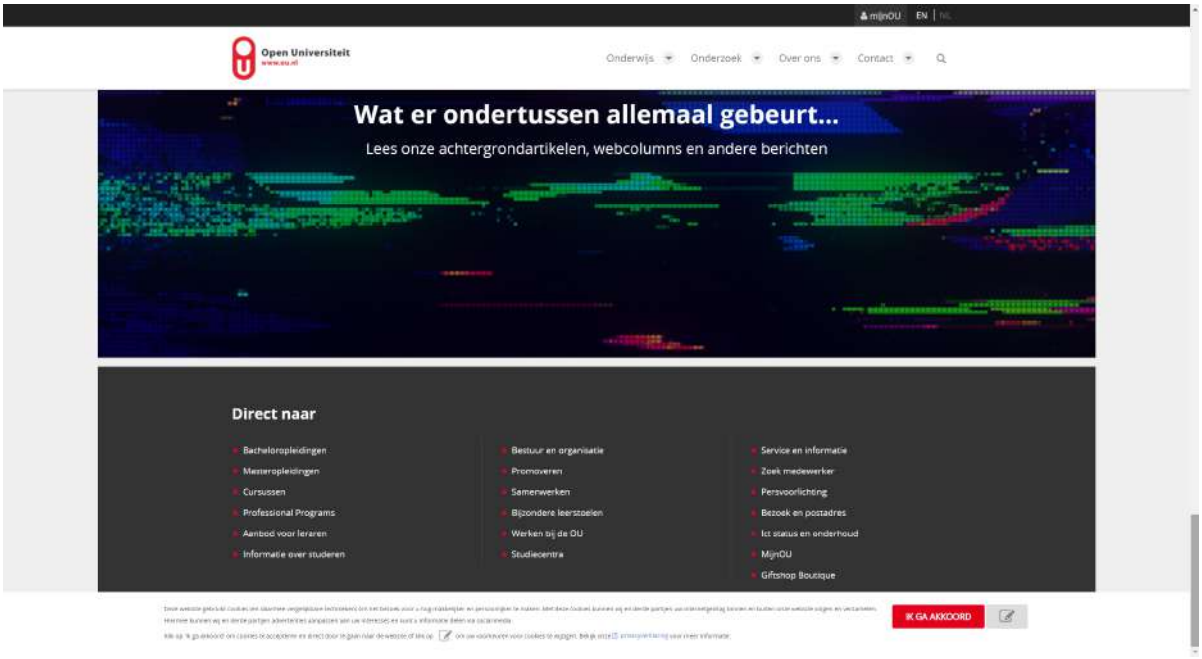


Figure 3.5: Website www.ou.nl

shown in figure 3.6. We need to take these incorrect results into account when we try to match the OCR result with the text retrieved via the accessibility API that we want to present to the user. We also noticed that the OCR results are individual words, as you can see in figure 3.3. All these words belong the the text at the bottom of figure 3.5. To match these results against the result from the webdriver protocol, we need to investigate how we can combine individual results into a sentence or even a paragraph. This will be investigated in research question 2a *How to combine the OCR results when the expected text consists out of multiple words?*

Figure 3.6: Widgets.

The next step was to retrieve the expected text from the accessibility API. To accomplish this, we used the default "*desktop generic protocol*" in combination with `Notepad` as SUT. By writing the "*Title*" property for all the widgets to the debug log, we expected to obtain all the visible text. We noticed that for some widgets the title was set but not visualized. We have highlighted two widgets in figure 3.6 for which the title was set, however not expected. For example, the "*UIAScrollbar*", based on the debug log we expected to see the text *Vertical*, but looking at figure 3.6, we can conclude that it did not visualize any text at all for the "*UIAScrollbar*". Also, for the "*UIAEdit*" we noticed that *Text Editor* was expected but as shown in the screenshot, there is no text at all for the "*UIAEdit*" widget. This, combined with the false positives of the OCR webdriver results, leads us to the following question.

How can we filter out the false positives of widgets that do not have a text that needs to be visualized?

The last step, to complete the proof of concept, is to compare the text retrieved from the accessibility API with the results from the OCR engine. The comparison results should be added to the report and affect the outcome of the test once we have detected a mismatch. We consider reporting the results to be part of the final implementation, but it is not in scope of the proof of concept. Matching the two results can not be achieved by performing a cross comparison look up. Both outcomes contain false positives and the OCR engine result contains single words. To solve this problem, we defined the following research question.
How to match the results from the OCR engine with the information that is extracted from the accessibility API?

3.3 Research questions

To come to a solution that will solve the problem stated in the project assignment 1.5, and with the findings gained by carrying out the proof of concept, we have identified the following research questions.

3.3.1 Main research question

How can OCR contribute to automatic validation that the text on GUI components, of an application under test by TESTAR, are entirely visible for the user?

3.3.2 Sub research questions

1. How can OCR help to automatically validate whether the text is entirely visible?
2. How can OCR be integrated into TESTAR in order to validate the visibility of the text for the application under test?
    - 2.1 How to combine the OCR results when the expected text consists out of multiple words?
    - 2.2 How can we filter out the false positives of widgets that do not have a text that needs to be visualized?
    - 2.3 How to match the results from the OCR engine with the information that is extracted from the accessibility API?
3. How can the implemented solution, for detecting automatically whether the text of GUI components is entirely visible, be validated?

Chapter 4

Design and implementation of a visual validation module

In this chapter, we will introduce our design for the visual validation module. Before we will explain each component within the new module in detail, we will first highlight the steps that TESTAR is taking while running a test. Based on this overview, we will introduce the new visual validation module and we explain how we have integrated the module in the current architecture. After this overview, we introduce a new settings framework. This new framework can run in parallel with the current settings framework. The primary reason to introduce this new framework is to make the introduction of the new components simpler. The new components will require a lot of configuration settings compared to the existing code. Because of the lack of flexibility, the current settings framework is not suitable for storing the configurations for the visual validation module.

4.1 Design overview

4.1.1 Current state of TESTAR

As we have mentioned in section 1.2.3, TESTAR can identify and execute actions on a SUT autonomously. In combination with the validation step afterwards, TESTAR is capable of testing an application without any user interaction required. In this subsection, we will elaborate on the high-level steps that TESTAR is taking while testing an application. There are multiple parameters that we can adjust when we want to test an application with TESTAR. We will highlight two of them:

1. The "*number of sequences*".
2. The "*number of actions*" a sequence is made up of.

By changing the "*number of sequences*", we specify how many times TESTAR should repeat testing the application before ending the test session. This does not mean that the performed actions are the same across different sequences.

With the "*number of actions*", the user can specify how many actions will be executed within a single sequence. An action can be for example clicking on a button, or providing text input.

The two concepts of "*sequences*" and "*actions*" are related to the "*outer loop*" and "*inner loop*" inside the codebase of TESTAR. When we start testing a SUT, we first enter the "*outer loop*", which will be executed for each sequence. The "*inner loop*" deals with all the action related code. In other words, the "*number of actions*" parameter specifies how many times the "*inner loop*" is being executed. The same relation is applicable for the "*outer loop*" and the "*number of sequences*" parameter.

Algorithm 1 shows an overview of the steps that are being taken by TESTAR when it is testing an application. We start each sequence by obtaining the initial state of the GUI. This takes place in the "*outer loop*". During this process, TESTAR creates a screenshot of the initial state of the application. From this point forward, the steps are related to the action. This means that we move to the "*inner loop*". It starts with deriving the available actions in the state. By inspecting all the elements, TESTAR can automatically determine which potential actions it can execute. From this list, TESTAR will select an action and execute it. Which action will be selected depends on the configured algorithm. Once TESTAR executes the action, it makes a screenshot of the element on which it has executed the action in isolation and obtains the new state. A "*verdict*" is being composed based upon the new state of the SUT. A "*verdict*" is the outcome of a test oracle. It determines whether the new state is erroneous. The severity of the "*verdict*" shall be used to determine whether TESTAR needs to abort the sequence or that it can proceed. Finally, another screenshot is being taken to capture the new state of the SUT. Once the number of specified actions has been executed by TESTAR and all the state verdicts are not marked as critical, the final sequence verdict shall be calculated. With this final verdict, the sequence has come to an end and a new sequence may be started depending on the number of defined sequences for this test.

Algorithm 1 Current TESTAR inner and outer loop

```
1: while numberOfSequences! = 0 do                          ▷ Outer loop
2:     getState()
3:     takeScreenshot()
4:     while numberOfActions! = 0 do                        ▷ Inner loop
5:         deriveAction()
6:         executeAction()
7:         takeActionScreenshot()
8:         getState()
9:         takeScreenshot()
10:        getVerdict()
11:        numberOfActions − −
12:    end while
13:    getFinalVerdict()
14:    numberOfSequences − −
15: end while
```

4.1.2 New design

For this research, we are going to investigate how we can extend the current capabilities of TESTAR with a module for visual validation. We introduce a new module named "*visual validation*" that contains all the code for this new feature. Figure 4.1 shows an architectural overview of the integration of the new module. When we enable the visual validation, every time that TESTAR

takes a screenshot from the SUT, we pause the normal flow and execute the visual validation (by means of calling "*analyzeScreenshot*"). On completion of the validation, the normal flow as described in Algorithm 1 will continue.

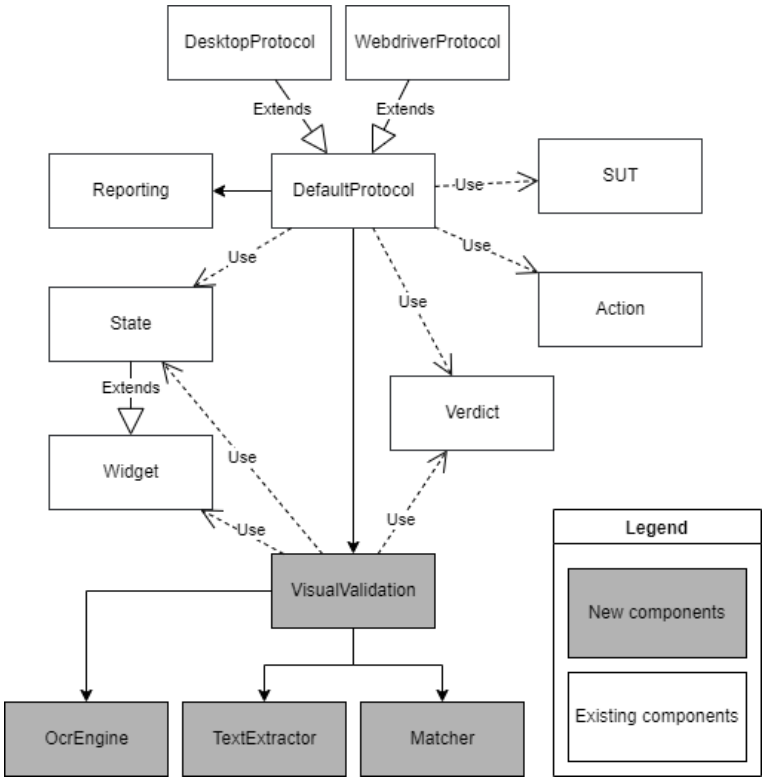


Figure 4.1: Architectural overview of the integration of the visual validation module.

Algorithm 2 shows the pseudo-code of "*analyzeScreenshot*" that is responsible for the visual validation. Algorithm 3 shows pseudo-code for integrating this. The new module comprises four components that are called from "*analyzeScreenshot*":

1. Extracting the text from the screenshot.
2. Gathering the expected text.
3. Matching the discovered text from the screenshot with the expected text.
4. Composing a verdict based on the matching results.

Algorithm 2 Pseudocode for analyzing the captured screenshot.

1: $identifiedText = extractText(screenshot)$
2: $expectedtText = obtainExpectedText(state)$
3: $result = matchText(identifiedText, expectedText)$
4: $updateVerdict(result, state)$;

Algorithm 3 Visual validation module integration pseudocode.

```
1: while numberOfSequences! = 0 do
2:     getState()
3:     takeScreenshot()
4:     analyzeScreenshot()                                  ▷ New code
5:     while numberOfActions! = 0 do
6:         deriveAction()
7:         executeAction()
8:         takeActionScreenshot()
9:         analyzeScreenshot()                              ▷ New code
10:        getState()
11:        takeScreenshot()
12:        analyzeScreenshot()                              ▷ New code
13:        getVerdict()
14:        numberOfActions − −
15:    end while
16:    getFinalVerdict()
17:    numberOfSequences − −
18: end while
```

Figure 4.2 shows a high-level class diagram of the new visual validation module. We will explain the OCR in section 4.3. Everything related to the OCR engine is coloured in medium shade of grey. The OCR engine is responsible for extracting the text from the screenshot. We will discuss the expected text in section 4.4. Expected text related components are coloured in the darkest grey colour. Expected text will be queried from the SUT and used as a reference. In section 4.5, we will elaborate on the matcher. The matcher components are coloured in the lightest shade of grey. The matcher will validate whether the expected text is visualised entirely by comparing the text with the OCR results.

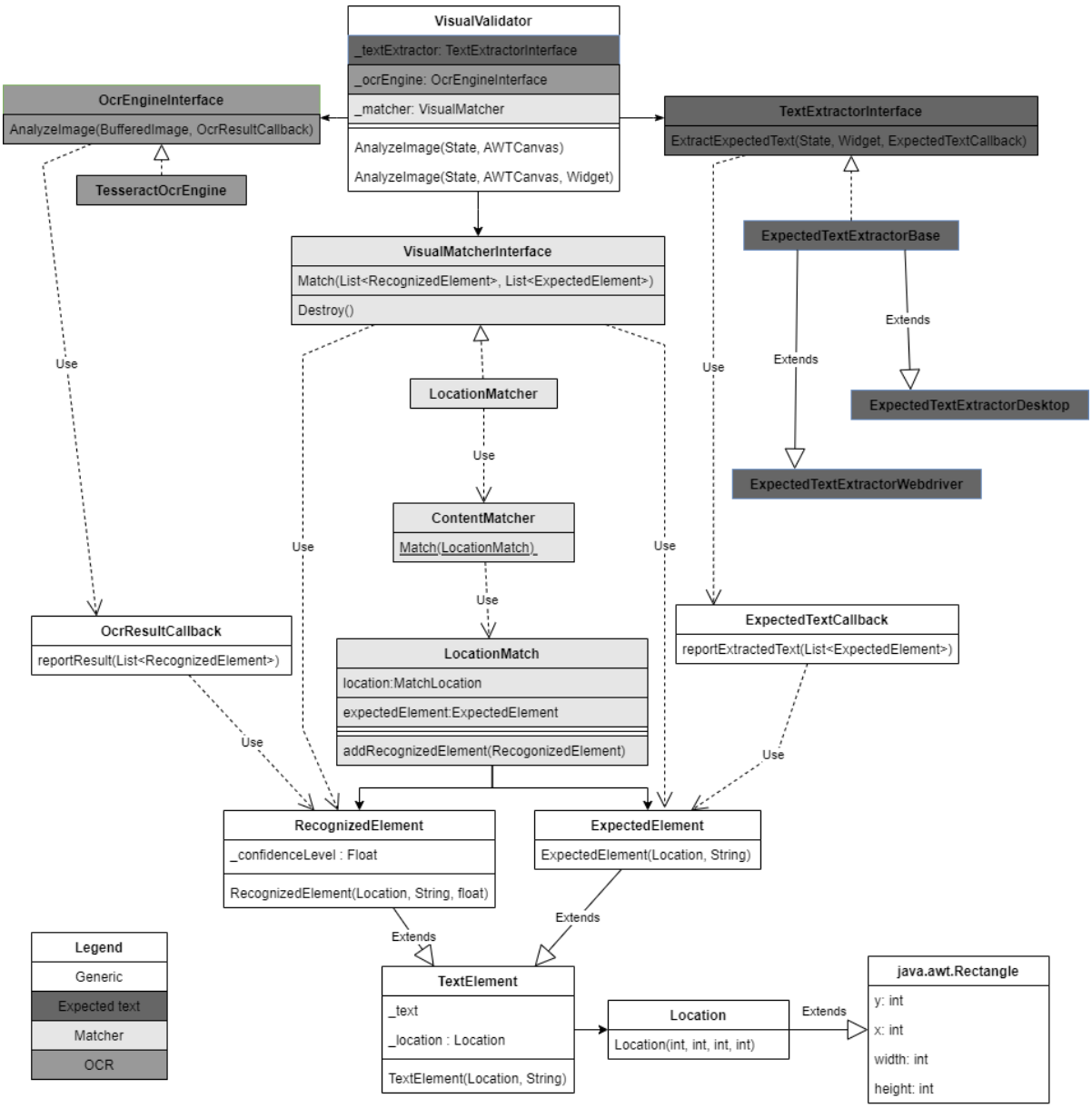


Figure 4.2: Class diagram of the visual validation module.

4.2 Settings framework

We have learned from the proof of concept in section 3.2, that extracting the text from a screenshot and matching the expected text requires support for exceptional cases. For example, while testing the Notepad application, the scrollbar, according to the accessibility API, the text *Vertical* was expected, while looking at the screenshot it became clear that the scrollbar had no text at all. Also, we have learned that the OCR engine returned most of the time individual words where the expected text was a sentence. Supporting these exceptional cases and optimizing the OCR engine requires an extendable settings framework.

The existing settings framework is based upon a text file with a custom key value structure. In general, the values consist out of a single element, extending this to support multiple elements would require additional development effort. If we would add the necessary settings to support

the visual validation module to the existing settings framework, it would affect the maintainability and usability in a negative way. All the exceptional cases that we need to include for the OCR engine would introduce a lot of clutter. However, the biggest downside of the existing implementation is that, when there is no settings file available, it creates a default file that contains all the available settings with their default values. This is one big monolith inside the codebase. Therefore, we have decided to design and implement a more flexible settings framework which can be used in parallel with the existing settings framework.

The new settings framework is an enabler for creating the visual validation module. It will make it possible to implement the solution that provides an answer to RQ2. We based the new settings framework upon XML and use the JAXB[1] package for interacting with the XML file. Figure 4.3 shows the class diagram of the new settings framework. The class "*ExtendedSettingsFile*" controls the read and write operations on the file. It only knows a single type, called "*RootSetting*". This class contains a list of "*Object*"s. When we want to read or write a particular setting, we try to match the given class with an entry from the list based on the class type. By doing so, we separate the specific settings structure from the generic interaction with the XML file. For reading from the XML file we use the "*UnMarshaller*" operation provided by JAXB, this does all the heavy lifting and provides a class with filled members when the read operation has succeeded. We annotated each specific setting class with "*@XmlRootElement*", this makes it possible for the "*UnMarshaller*" to map the XML tags on the attributes of the class and vice versa when using the "*Marshaller*". The settings framework can be used by first calling the "*Initialize*" function of the "*ExtendedSettingsFactory*". Second, a new settings class needs to be created. This new settings class needs to extend the "*ExtendedSettingBase*" and use the previously described annotations, and have a static constructor which initializes the members with a default value. Once this is implemented and "*Initialize*" is called, the "*createSettings*" from the factory class can be used to create the setting. If the setting already exists in the file, the new class will be filled with the stored data, otherwise the default values will be written to the file. The factory will create a dedicated "*ExtendedSettingContainer*" for the new setting. This container provides the link between the "*ExtendSettingsFile*" and the new setting class. If the factory receives the instruction to save all the data, it will instruct all the containers to write the data to the file.

[1] https://www.oracle.com/technical-resources/articles/javase/jaxb.html

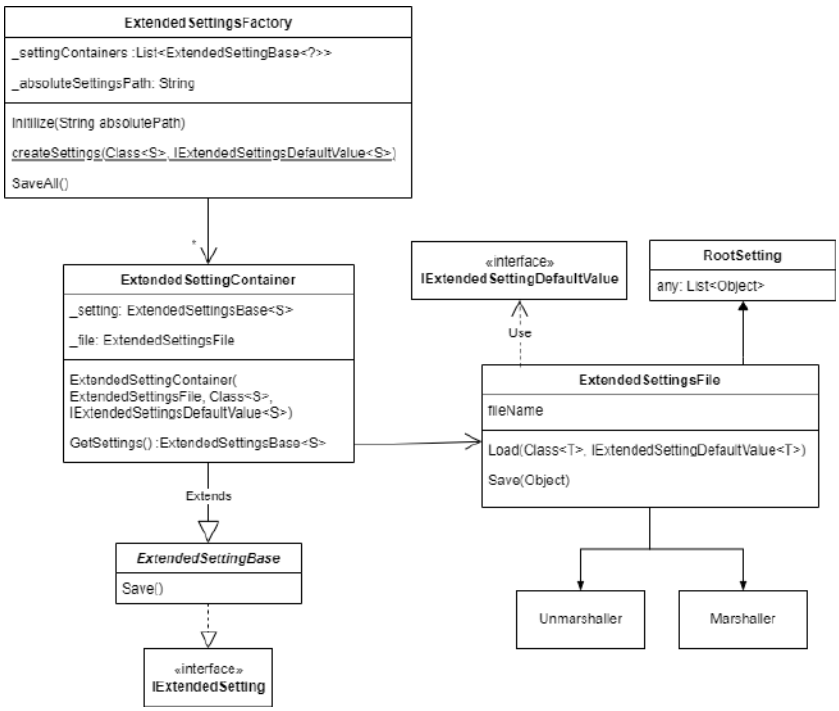


Figure 4.3: Class diagram of the new settings framework.

4.3 OCR

In our solution to answer the research questions, we have created an abstraction layer for the OCR engine integration. By doing so, it contributes to flexibility, and it becomes relatively easy to replace the integration. This can be achieved by implementing a new dedicated OCR engine class which implements the "*OcrEngineInterface*", which is listed in figure 4.2. The abstraction layer contributes in a positive way to experiment with other OCR engines when needed. By executing the proof of concept, we have gained a lot of experience with OCR engines. Based on the outcome of the selection process 2.2, we have integrated Tesseract for our implementation. With the output from the OCR engine, we obtain the actual visualization of the text. This will be used as input to determine if the SUT contains a TPF. We have identified that the OCR engine sometimes finds characters where the SUT actually has images or icons positioned. For example, during the experiment with the Notepad application, the OCR engine identified the close button as the character 'X'. For this integration, we have chosen to configure the OCR engine to return single words. This allows us to process small chunks when we need to match them with the expected text. The downside is that the OCR result does not include whitespaces like tab and space. We have also learned from the proof of concept that the OCR engine needs a significant amount of time to inspect the image before it returns all the identified text elements. Therefor, we have implemented a dedicated thread for analyzing the image. This enables us to analyze the screenshot in the background while TESTAR has time to execute other tasks. This is, however, somewhat limited. The outcome of the visual validation affects the verdict of the executed action. In our implementation, we start collecting the expected text while the OCR engine analyzes the screenshot. Once the OCR engine is finished with processing the screenshot, we will convert the specific OCR results into a list of "*RecognizedElement*". This class holds the identified text, the location of the identified text relative to the screenshot, and the confidence level of the OCR engine.

For the first test with the integrated OCR engine, we have used the "*desktop protocol*" in combination with the Notepad application. We have changed the implementation compared to the proof of concept 3.2. For the proof of concept, we used the screenshot which is stored on disk. Instead of writing the screenshot first to disk before we read it back and feed it to the OCR engine, we create a copy of the raw buffer and feed the copy directly to the OCR engine. This saves a significant amount of time because we do not need to wait for the image to be written to disk. Figure 4.4 shows the screenshot of the SUT which has been analyzed by the OCR engine. Table 4.1 shows the outcome of the OCR engine. After performing a manual inspection, we concluded that we need to filter out the results 01, 02, 14 and the toolbar icons; 06, 07, 08. The remaining OCR results are identified correctly.

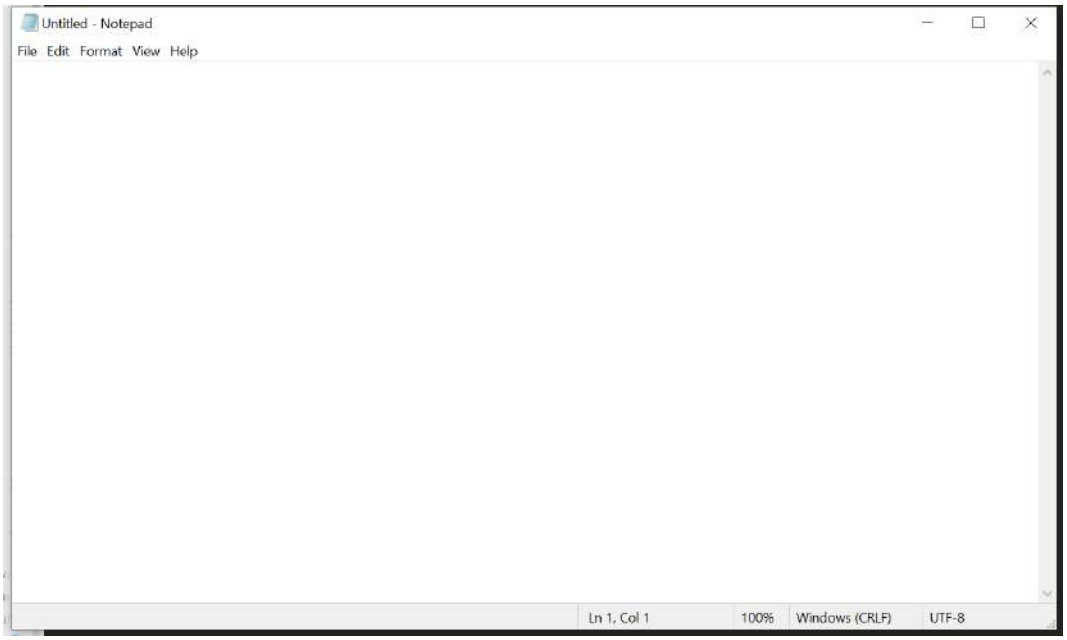


Figure 4.4: Screenshot of the desktop application used for the first OCR result analysis.

Result no.	Text	Confidence level	Location
01	”	95.0	x=11,y=1,width=1,height=841
02	’|’	9.904564	x=41,y=12,width=6,height=23
03	’Untitled’	93.04931	x=53,y=17,width=60,height=14
04	’-’	92.701096	x=120,y=25,width=5,height=1
05	’Notepad’	96.42946	x=132,y=17,width=68,height=18
06	’-’	10.836578	x=1238,y=23,width=15,height=1
07	’o’	76.49873	x=1307,y=16,width=15,height=15
08	’x’	92.79747	x=1378,y=16,width=15,height=15
09	’File’	96.02964	x=20,y=54,width=25,height=14
10	’Edit’	96.02964	x=60,y=54,width=29,height=14
11	’Format’	96.18999	x=104,y=55,width=56,height=13
12	’View’	95.592995	x=174,y=55,width=37,height=13
13	’Help’	96.32465	x=226,y=54,width=35,height=18
14	”	95.0	x=1416,y=0,width=13,height=850
15	’Ln’	75.38964	x=790,y=818,width=16,height=13
16	’1,’	75.38964	x=814,y=818,width=11,height=15
17	’Col’	79.91156	x=831,y=817,width=25,height=14
18	’1’	79.91156	x=864,y=818,width=5,height=13
19	’100%’	95.61263	x=996,y=818,width=43,height=13
20	’Windows’	46.88897	x=1069,y=817,width=74,height=14
21	’(CRLF)’	96.78083	x=1149,y=818,width=47,height=16
22	’UTF-8’	14.203491	x=1250,y=818,width=46,height=13

Table 4.1: First OCR results of the Notepad application.

However, when we selected the “*webdriver protocol*”, the OCR results did not meet the expectations. Figure 4.5 shows the screenshot of the website that we have used to verify the OCR implementation for a website. We listed the results of the OCR engine in table 4.2. The majority of OCR results consisted out of garbage characters and none of the results could be found in the screenshot. During the investigation of why the results between the two different protocols differ, we noticed that TESTAR uses different techniques to acquire a screenshot:

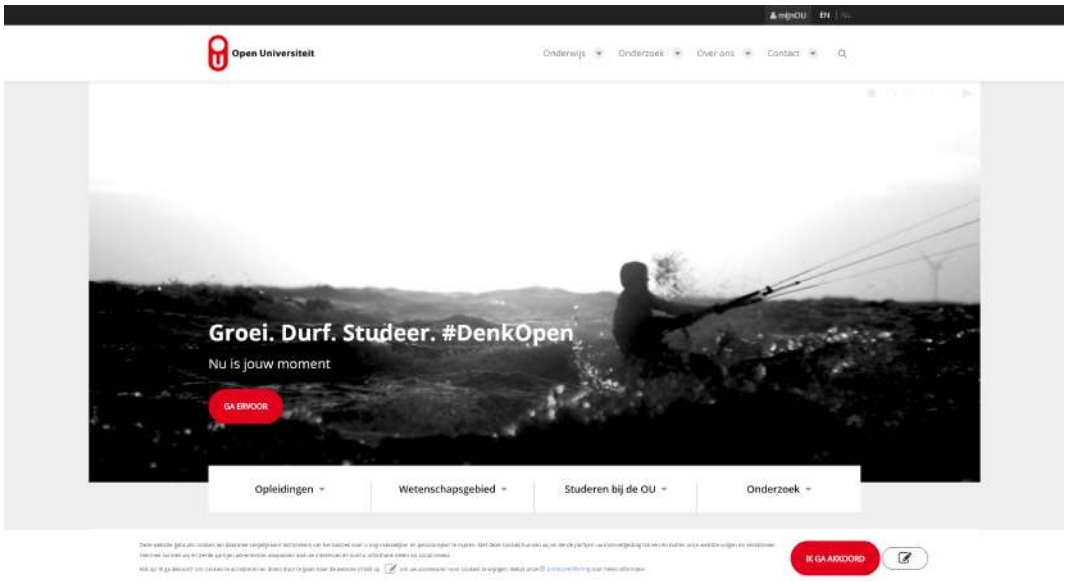


Figure 4.5: Screenshot of the `www.ou.nl` website.

- For the "*desktop protocol*", TESTAR uses the screenshot capabilities of the "*Robot*" class inside the AWT package. The function call "*createScreenCapture*" returns a "*BufferImage*" in the "*TYPE_INT_ARGB*" format.

- For the "*webdriver protocol*", TESTAR instructs the Selenium webdriver to take screenshot. Once completed, a "*BufferImage*" is returned in the "*TYPE_4BYTE_ABGR*" format.

Result no.	Text	Confidence level	Location
01	'Ee<VO™'	0.8526306	x=0,y=1,width=2854,height=110
02	'a'	0.0	x=0,y=111,width=1481,height=111
03	'ee'	27.88868	x=0,y=222,width=2854,height=111
04	'SSS]'	30.875565	x=0,y=333,width=2854,height=111
05	'————————'	10.487862	x=0,y=444,width=2854,height=110
06	'Se'	35.42514	x=0,y=554,width=2854,height=111
07	'———'	0.0	x=0,y=591,width=410,height=189
08	'SEE'	45.13797	x=0,y=665,width=2854,height=111
09	'SS'	45.492805	x=0,y=702,width=394,height=189
10	'————————'	0.0	x=0,y=887,width=2854,height=110
11	'————————'	0.0	x=0,y=924,width=486,height=188
12	'a'	15.075256	x=2315,y=1108,width=536,height=111
13	'VS'	7.975235	x=299,y=1145,width=142,height=189
14	'————————'	7.975235	x=0,y=1219,width=2854,height=111
15	'RN'	26.295235	x=0,y=1330,width=2854,height=110
16	'ne'	9.02877	x=0,y=1437,width=308,height=132
17	'SEE'	54.873844	x=0,y=1441,width=2854,height=128

Table 4.2: OCR results webdriver protocol from raw image buffer.

When we feed the "*TYPE_4BYTE_ABGR*" format directly to the OCR engine, garbage is being returned. To solve this, we have introduced a workaround and made a small adjustment to the configuration of the OCR engine. By first saving the buffer to disk and using the recommended "*pixRead*" function from the leptonica library, we can feed the image in such a way that the engine can process the given image. Also, by explicitly setting the page segmentation mode of the Tesseract engine to "*PSM_AUTO_OSD*", the outcome of the OCR engine does not contain any garbage results anymore. Table 4.3 shows the results of the OCR engine, including the adjustments, after analyzing the same input image 4.5.

Result no.	Text	Confidence level	Location
001	''	95.0	x=2058,y=14,width=104,height=27
002	''	95.0	x=2812,y=0,width=40,height=58
003	'Open'	96.325	x=614,y=115,width=60,height=33
004	'Universiteit'	96.16965	x=673,y=115,width=161,height=33
005	'Onderwijs'	90.38119	x=1453,y=120,width=111,height=24
006	'Â»'	72.17652	x=1595,y=126,width=13,height=7
007	'Onderzoek'	63.30056	x=1656,y=120,width=120,height=18
008	'Â» '	27.155235	x=1807,y=126,width=13,height=7
009	'Overons'	73.06729	x=1868,y=121,width=97,height=17
010	'Â»Â¥'	0.0	x=1997,y=126,width=12,height=7

011	'Contact'	92.02052	x=2057,y=121,width=83,height=17
012	'v'	91.021835	x=2171,y=126,width=13,height=7
013	'a'	49.714867	x=929,y=803,width=65,height=13
014	'Groei.'	92.70035	x=553,y=857,width=172,height=47
015	'Durf.'	92.0988	x=749,y=856,width=143,height=48
016	'Studeer.'	91.90916	x=914,y=830,width=246,height=84
017	'#De'	92.51841	x=1180,y=860,width=115,height=44
018	'Nu'	96.34135	x=554,y=952,width=43,height=26
019	'is'	96.28991	x=611,y=951,width=23,height=27
020	'jouw'	96.29336	x=642,y=951,width=83,height=36
021	'moment'	96.29336	x=738,y=955,width=140,height=23
022	''	95.0	x=224,y=679,width=2408,height=609
023	'Opleidingen'	92.41849	x=677,y=1287,width=154,height=35
024	'Wetenschapsgebied'	92.29064	x=1051,y=1287,width=270,height=35
025	'Studeren'	91.24835	x=1511,y=1291,width=115,height=21
026	'bij'	96.51697	x=1638,y=1291,width=27,height=27
027	'de'	95.97179	x=1676,y=1291,width=29,height=21
028	'OU'	96.84964	x=1716,y=1293,width=37,height=19
029	'Onderzoek'	92.91361	x=2002,y=1291,width=140,height=21
030	''	95.0	x=373,y=1450,width=39,height=8
031	''	95.0	x=808,y=1448,width=24,height=12
032	'Deze'	96.80641	x=367,y=1454,width=7,height=4
033	'website'	93.18645	x=414,y=1450,width=39,height=8
034	'gebruikt'	89.7969	x=458,y=1447,width=55,height=15
035	'cookies'	96.822914	x=518,y=1450,width=47,height=8
036	'(en'	93.3004	x=572,y=1437,width=19,height=31
037	'daarmee'	81.286964	x=598,y=1447,width=57,height=11
038	'vergelijkbare'	79.84029	x=661,y=1448,width=85,height=14
039	'techniek:'	75.67718	x=751,y=1447,width=55,height=11
040	'ek'	60.52777	x=917,y=1450,width=12,height=8
041	'voor'	92.74515	x=931,y=1437,width=28,height=31
042	'u'	92.766	x=971,y=1450,width=6,height=8
043	'nog'	93.29224	x=986,y=1450,width=21,height=12
044	'makkelijker'	83.65679	x=1013,y=1447,width=74,height=11
045	'en'	93.17332	x=1088,y=1437,width=12,height=31
046	'persoonijker'	33.78195	x=1115,y=1447,width=79,height=15
047	'te'	96.54854	x=1206,y=1450,width=11,height=8
048	'maken.'	95.68714	x=1224,y=1450,width=42,height=8
049	'Met'	96.94938	x=1277,y=1448,width=24,height=10
050	'deze'	96.96833	x=1307,y=1447,width=29,height=11
051	'cookies'	94.54039	x=1342,y=1450,width=48,height=8
052	'kunnen'	91.49214	x=1396,y=1447,width=48,height=11
053	'wij'	81.15932	x=1452,y=1450,width=15,height=12
054	'en'	81.15932	x=1473,y=1450,width=14,height=8
055	'derde'	93.071976	x=1494,y=1447,width=37,height=11
056	'partijen'	82.90667	x=1538,y=1449,width=48,height=9
057	'uw'	93.16978	x=1593,y=1450,width=18,height=8
058	'internetgedrag'	84.5601	x=1617,y=1450,width=98,height=12
059	'binnen'	92.34654	x=1723,y=1450,width=42,height=8
060	'en'	93.22515	x=1771,y=1450,width=14,height=8
061	'buiten'	91.98418	x=1792,y=1448,width=41,height=10

062	'onze'	96.96304	x=1839,y=1450,width=30,height=8
063	'website'	96.92143	x=1875,y=1449,width=50,height=9
064	'volgen'	96.89455	x=1930,y=1450,width=42,height=12
065	'en'	93.296844	x=1979,y=1450,width=14,height=8
066	'verzamelen.'	91.69501	x=2000,y=1448,width=75,height=10
067	''	95.0	x=453,y=1469,width=52,height=21
068	''	95.0	x=774,y=1448,width=148,height=53
069	'Hiermee'	87.283005	x=367,y=1476,width=55,height=10
070	'kun'	70.57867	x=428,y=1475,width=22,height=11
071	'en'	93.29324	x=505,y=1478,width=14,height=8
072	'derde'	93.0686	x=525,y=1475,width=38,height=11
073	'partijen'	88.92763	x=569,y=1477,width=49,height=13
074	'advertenties'	91.082855	x=624,y=1475,width=82,height=11
075	'aanpasse!'	45.996178	x=712,y=1478,width=61,height=11
076	'n'	45.627686	x=923,y=1478,width=4,height=8
077	'kunt'	93.266464	x=933,y=1475,width=89,height=11
078	'u'	90.6978	x=960,y=1465,width=11,height=31
079	'inform:'	92.77951	x=973,y=1475,width=49,height=11
080	''	95.0	x=1024,y=1475,width=36,height=11
081	'elen'	26.299805	x=1062,y=1478,width=26,height=8
082	'via'	96.694016	x=1094,y=1478,width=17,height=8
083	'social'	96.694016	x=1117,y=1478,width=32,height=8
084	'media'	96.350975	x=1159,y=1478,width=39,height=8
085	''	95.0	x=1280,y=1511,width=50,height=12
086	''	95.0	x=2360,y=1454,width=131,height=69
087	'psite'	73.299774	x=917,y=1513,width=30,height=10
088	'of'	96.586334	x=953,y=1512,width=13,height=11
089	'klik'	67.51427	x=971,y=1513,width=20,height=10
090	'op'	67.51427	x=997,y=1515,width=14,height=11
091	'|'	90.3918	x=1017,y=1506,width=16,height=24
092	'@#'	40.95728	x=1042,y=1510,width=11,height=10
093	'om'	88.74022	x=1067,y=1515,width=20,height=8
094	'uwvoorkeuren'	87.434875	x=1093,y=1515,width=99,height=8
095	'voor'	96.57351	x=1199,y=1515,width=28,height=8
096	'cookies'	96.85126	x=1233,y=1512,width=47,height=11
097	''	95.0	x=851,y=1512,width=65,height=11
098	'Klik'	90.296135	x=367,y=1513,width=21,height=10
099	'op'	90.296135	x=394,y=1515,width=15,height=8
100	'"Ik'	75.91874	x=418,y=1512,width=9,height=11
101	'ga'	93.25461	x=433,y=1515,width=14,height=12
102	'akkoordâ€™'	61.234543	x=453,y=1512,width=56,height=11
103	'om'	96.95177	x=515,y=1515,width=19,height=8
104	'cookies'	88.12414	x=541,y=1515,width=48,height=8
105	'te'	93.08019	x=594,y=1515,width=12,height=8
106	'accepteren'	93.08019	x=612,y=1514,width=72,height=12
107	'en'	96.9013	x=690,y=1515,width=15,height=8
108	'direct'	96.740776	x=711,y=1512,width=37,height=11
109	'door'	96.63357	x=753,y=1512,width=24,height=11
110	'te'	96.91228	x=788,y=1515,width=12,height=8
111	'gaan'	85.00469	x=805,y=1515,width=31,height=12
112	'n.'	54.914795	x=843,y=1515,width=5,height=2

113	'igen.'	89.20632	x=1332,y=1515,width=23,height=12
114	'Bekijk'	96.39179	x=1365,y=1512,width=37,height=11
115	'onze'	95.28555	x=1408,y=1502,width=30,height=30
116	'[3'	40.41153	x=1438,y=1502,width=17,height=30
117	'privacyverklaring'	86.60008	x=1465,y=1512,width=112,height=14
018	'voor'	96.282135	x=1583,y=1515,width=28,height=8
119	'meer'	92.99922	x=1619,y=1515,width=25,height=8
120	'informatie.'	82.87619	x=1656,y=1512,width=68,height=11

Table 4.3: OCR results webdriver protocol with saved image.

4.4 Expected text

For the implementation of the expected text extraction class, we have first tested with the desktop application *Notepad*. We have started by querying the expected text via the selected "*desktop protocol*". We accomplish this by iterating over the "*widget tree*". The "*widget tree*" is composed for each "*State*" of the SUT. Depending on the selected protocol, a dedicated "*StateBuilder*" will generate the "*widget tree*". This "*StateBuilder*" can request detailed information about the SUT from the system. The "*widget tree*" is a tree hierarchy, starting with the root widget of the SUT. The "*widget tree*" stores relevant information for each widget, e.g.:

- Unique identifier.
- The ancestor path.
- The role, this contains the type of widget, e.g. a button.
- The dimensions
- The location.

This information can be queried by using the dedicated "*Tags*". Testar uses common "*Tags*" for information which is independent of the selected protocol, e.g. "*Role*". Depending on the selected protocol, custom "*Tags*" are used, e.g.:

- For the "*desktop protocol*", there is a "*Tag*" "*UIAIsEnabled*". If the value for this "*Tag*" is true, the widget is enabled.
- For the "*webdriver protocol*", there is a "*Tag*" "*WebHasKeyboardFocus*". If the value for this "*Tag*" is true, the widget has the focus.

While iterating over all the available widgets after a screenshot has been taken, we obtained their value by requesting the tag "*Title*". Table 4.4 shows an overview of the expected text for the Notepad application as shown in figure 4.4. Based on the proof of concept, we already knew that this was not enough, and that for special cases we need to use a different tag instead of the "*Title*" tag. So we have created a dedicated setting for this inside the newly implemented settings framework 4.2.

Result no.	Text	Location
01	'Untitled - Notepad'	x=0,y=0,width=1429,height=850
02	'Text Editor'	x=12,y=75,width=1408,height=729
03	'Status Bar'	x=12,y=804,width=1408,height=36
04	'Application'	x=12,y=45,width=1408,height=28
05	'Vertical'	x=1393,y=75,width=16,height=729
06	' Ln 1, Col 1'	x=777,y=807,width=208,height=34
07	' 100%'	x=987,y=807,width=73,height=34
08	' Windows (CRLF)'	x=1062,y=807,width=178,height=34
09	' UTF-8'	x=1242,y=807,width=154,height=34
10	'System'	x=12,y=12,width=33,height=33
11	'Minimize'	x=1209,y=1,width=70,height=43
12	'Maximize'	x=1279,y=1,width=70,height=43
13	'Close'	x=1350,y=1,width=70,height=43
14	'File'	x=12,y=45,width=40,height=28
15	'Edit'	x=51,y=45,width=43,height=28
16	'Format'	x=96,y=45,width=70,height=28
17	'View'	x=166,y=45,width=51,height=28
18	'Help'	x=217,y=45,width=51,height=28
19	'Line up'	x=1393,y=75,width=16,height=25
20	'Line down'	x=1393,y=778,width=16,height=25
21	'System'	x=12,y=12,width=33,height=33

Table 4.4: Expected text desktop protocol unfiltered

We have identified four scenarios for obtaining the expected text for a desktop application. Each scenario is supported by the new settings framework and can be adjusted by modifying the settings XML file.

The first scenario that we identify as the default scenario uses the "*Title*" tag to obtain the expected text as described above.

The second scenario applies to cases where we need to use a different tag based on the role of the widget. Each widget has a role that can be obtained from the "*widget tree*". This role states the type of widget, e.g. *TextEdit*, *Button* widget. For this second scenario we use the settings framework to specify, for a particular role which tag must be queried in order to obtain the right expected text. For example, for the *TextEdit* widget we need to query the "*UIAValueValue*" tag. Figure 4.6 shows the XML snippet, used by the settings framework, for this scenario.

```
<widget>
    <role>UIAEdit</role>
    <tag>UIAValueValue</tag>
    <ignore>false</ignore>
    <ancestor></ancestor>
</widget>
```

Figure 4.6: XML snippet of using a different tag to obtain the expected text for a widget.

The third scenario covers the widgets that we want to ignore based on their role. We have noticed during the proof of concept that, for example, for the scrollbar widget the text *Scrollbar* was expected when the "*Title*" tag is queried. However, from visual inspection we know that there is no text shown at all for the scrollbar widget.

The fourth scenario is a more advanced exclusion compared to the third scenario. In this case, we want to ignore a particular widget based on its role in a specific place. We can locate a widget by inspecting their ancestors. The application icon in the menu bar is identified by the role of *MenuItem*, in most cases this returns the actual expected text. For example, *File*, *Setting* or *Help*, but when the *MenuItem* contained the specific ancestor path of *::UIAMenuBar::UIATitleBar::UIAWindow::Process*, it was showing the application icon in the upper left corner of the SUT instead of a text. Figure 4.7 shows the XML snippet for this scenario.

```
<widget>
    <role>UIAMenuItem</role>
    <tag></tag>
    <ignore>true</ignore>
    <ancestor>::UIAMenuBar::UIATitleBar::UIAWindow::Process</ancestor>
</widget>
```

Figure 4.7: XML snippet of ignoring a widget based on its ancestor path.

As mentioned in the previous section, retrieving the expected text is done in parallel with the OCR operation, despite of the fact that extracting the text is relatively fast compared to the OCR procedure. By placing this in a thread, we created the possibility to run these two procedures in parallel and reduced the required amount of time to test the application. Retrieving the expected text is accomplished by examining the current "*State*" of the SUT. The "*State*" reflects the current state of the SUT and contains all the widgets stored as a tree. While iterating over all the widgets in the "*State*", we consult the configuration to determine the right approach to process the widget. The expected text will serve as the reference input for identifying TPFs in the SUT.

Besides the expected text, we also obtain the position of the widget. We do this by querying the "*Shape*" of each widget. This returns the absolute location of the widget on the screen. By obtaining the absolute location of the root window of the SUT, we can calculate the relative position of the widget. This is needed because the screenshot is aligned to only capture the SUT, so the OCR results consist of relative positions.

Table 4.5 shows an overview of the expected text for the Notepad application. We have composed this overview by applying the filters based on the four different scenarios. Listing A.1 contains the entire XML, used by our new settings framework to define the filters. These expected text elements are obtained while the application was in the state as shown in figure 4.4.

Result no.	Text	Location
01	'Untitled - Notepad'	x=0,y=0,width=1429,height=850
02	' Ln 1, Col 1'	x=777,y=807,width=208,height=34
03	' 100%'	x=987,y=807,width=73,height=34
04	' Windows (CRLF)'	x=1062,y=807,width=178,height=34
05	' UTF-8'	x=1242,y=807,width=154,height=34
06	'File'	x=12,y=45,width=40,height=28
07	'Edit'	x=51,y=45,width=43,height=28
08	'Format'	x=96,y=45,width=70,height=28
09	'View'	x=166,y=45,width=51,height=28
10	'Help'	x=217,y=45,width=51,height=28

Table 4.5: Expected text desktop protocol filtered

Now that we have obtained the correct expected text and we are able to make changes via the settings framework, for our implementation for a desktop application, we shift the focus towards the "*webdriver protocol*". With the "*webdriver protocol*" selected, TESTAR can analyze and test websites. The first difference we noticed while using a "*webdriver protocol*", was that the "*Tags*" are not the same compared to the "*desktop protocol*". To obtain the correct text for elements shown on the website, we needed to use the tag "*WebTextContent*" instead. Table 4.4 shows the expected text when using the appropriate tag to obtain the expected text. After comparing the expected text elements with the screenshot in image 4.5, we noticed that the expected text results contained entries that are not visible on the screenshot. After investigating this discrepancy, we concluded that the expected text result contained the text elements of the entire website. In order to solve this, we had to distinguish between the visible elements and the elements that are located off screen, when extracting the expected text. By incorporating the check whether the tag "*WebIsFullOnScreen*" is true, we filter out the elements that are not visible. However, result 001 from table A.1, in the appendix, is still included in the expected text results. The widget is positioned in the upper left corner of the website and therefore it is correctly marked "*WebIsFullOnScreen*". However, looking at the screenshot, we can not find the expected text `Naar content`. After inspecting the source code, we noticed that the text element was clipped entirely. This means that the rectangle which contains the text was completely reduced. In other words, there were no pixels available which could visualize the text. In order to ignore this text element, we have added a dedicated filter to the settings file to ignore this widget based on its ancestors. This filter, together with the additional check if the widget is visible on the screen, we have filtered out all the false positives for the expected text. Table 4.6 shows the filtered result of the expected text elements for image 4.5.

Result no.	Text	Location
01	'mijnOU'	x=2047,y=0,width=126,height=55
02	'Deze website gebruikt cookies (en daarmee vergelijkbare technieken) om het bezoek voor u nog makkelijker en persoonlijker te maken. Met deze cookies kunnen wij en derde partijen uw internetgedrag binnen en buiten onze website volgen en verzamelen. Hiermee kunnen wij en derde partijen advertenties aanpassen aan uw interesses en kunt u informatie delen via social media. Klik op 'Ik ga akkoord' om cookies te accepteren en direct door te gaan naar de website of klik opom uw voorkeuren voor cookies te wijzigen. Bekijk onze voor meer informatie.'	x=364,y=1438,width=1714,height=103
03	'privacyverklaring'	x=1441,y=1506,width=133,height=18
04	'Ik ga akkoord'	x=2158,y=1474,width=157,height=27
05	'Onderwijs'	x=1428,y=54,width=201,height=150
06	'Onderzoek'	x=1630,y=54,width=211,height=150
07	'Over ons'	x=1842,y=54,width=189,height=150
08	'Contact'	x=2032,y=54,width=174,height=150
09	'en'	x=2191,y=10,width=48,height=27
10	'nl'	x=2248,y=10,width=34,height=27
11	'Opleidingen'	x=549,y=1239,width=438,height=126
12	'Wetenschapsgebied'	x=988,y=1239,width=438,height=126
13	'Studeren bij de OU'	x=1426,y=1239,width=438,height=126
14	'Onderzoek'	x=1866,y=1239,width=438,height=126
15	'Groei. Durf. Studeer. #DenkOpen'	x=549,y=840,width=1228,height=100
16	'Nu is jouw moment'	x=549,y=940,width=1228,height=67
17	'Ga ervoor'	x=549,y=1032,width=199,height=93

Table 4.6: Expected text webdriver protocol filtered

4.5 Matching results

Matching the expected text with the OCR results is not as straightforward as it may sound. Because of the detection errors made by the OCR engine, it is not possible to match the detected text by the OCR engine directly with the expected text. These detection errors can vary from a minor mistake like, for example, an incorrect letter case, till a major mistake like a ghost character that the OCR engine should not detect in the first place. We use the location of the expected text, and the discovered text for the matcher algorithm. The base principal of our matching logic is that we determine whether the discovered text intersects with the surface area of the expected text. We calculate the surface area based on the dimension and location of the text element. If the surface area of the discovered text intersects with an expected text

element, we create a relation between the two text components so that in a later stage we can compare the content of the text elements. Figure 4.8 shows the bounding box of the expected text. The bounding box is visualized in red based on the location and the dimensions of the widget. The widget is located at coordinate (208,47) and is 52 pixels width and has a height of 25 pixels. Figure 4.9 shows the bounding box of the detected text by the OCR engine in blue. The OCR result is detected at coordinate (217,51) and has a width of 34 pixels and a height of 19 pixels. Figure 4.10 shows the two bounding boxes combined. Here we can see that the surface area of the expected text and the discovered text intersect. Therefore, these two are linked together by our matching algorithm.

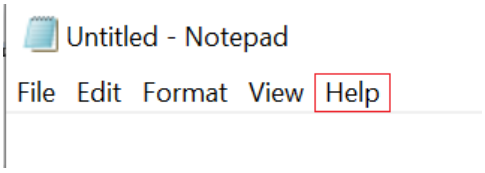


Figure 4.8: Bounding box of expected text.

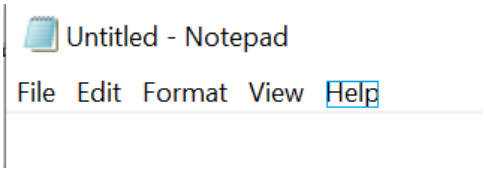


Figure 4.9: Bounding box of the ocr result.

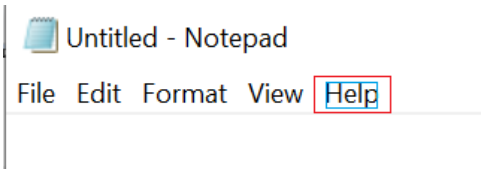


Figure 4.10: Intersecting elements.

The location property contains the relative X and Y position on the screen, as well as the width and height. We used the expected text as our baseline and try to link potential OCR results based on their location. To obtain the location for the expected text, we queried the selected protocol. However, we could not use the obtained location straight away. The problem was that the returned location reflects the dimension and location of the widget itself and not the location of the actual text. In most cases, this was just an offset of a couple of pixels, except for the title bar. We have chosen not to modify the returned location of the widget containing the expected text. In fact, we even benefit from a slightly larger surface area. This increases the change that we can find a match with the OCR results.

When we requested the location for the title bar, the result reflected the entire surface of the application window. If we did not anticipate on this, the algorithm would mark all the OCR results as potential matches. This was not a problem on its own. However, we decided to optimize the implementation of the matcher to speed up the process. We have done this by only matching the text elements which have not been matched before. Algorithm 4 shows the pseudocode for the optimization. By doing this, the list with unmatched text elements will shrink with every match. We have chosen for the most simplistic approach, where we iterated over all the OCR results every time we tried to find matches for an entry of the expected text result. By removing the OCR entry from the OCR results list when it lays inside the location, we improved the performance of the matching algorithm. However, if we did not resolve the title bar problem, it resulted in a scenario where the algorithm had linked all the OCR results to the title bar and that all the other expected text elements remained unlinked.

Algorithm 4 Optimization of the matcher algorithm

```
1: getRecognizedText()
2: getExpectedText()
3: while index! = expextedTextElements.size() do
4:     expextedText = expextedTextElements[index]
5:     intersectedText = getIntersectedText(expectedText, recognizedTextElements)
6:     if intersectedText! = null then
7:         result.addLocationMatch(expextedText)
8:         result.matchContent(expecetdText, intersectedText)
9:         recognizedTextElements.remove(intersectedText)
10:    else
11:        result.addNoLocationMatch(expextedText)
12:    end if
13:    index + +
14: end while
```

We have solved this problem by sorting the expected elements based on their surface area. Algorithm 5 shows the matcher algorithm, including the sorting step. The matching algorithm starts with matching the expected text elements with the smallest surface area first and removed them from the remaining list. We eventually ended up with the largest surface areas. Because we had already matched the smaller surface areas, we have discovered text elements by the OCR engine which are not matched yet, and therefore actually belong to the expected text elements with a larger surface area.

Algorithm 5 Optimization and of the matcher algorithm

```
1: getRecognizedText()
2: getExpectedText()
3: sortExpectedText(expextedTextElements)          ▷ Sorted based on the size of the surface area
4: while index! = expextedTextElements.size() do
5:     expextedText = expextedTextElements[index]
6:     intersectedText = getIntersectedText(expectedText, recognizedTextElements)
7:     if intersectedText! = null then
8:         result.addLocationMatch(expextedText)
9:         result.matchContent(expecetdText, intersectedText)
10:        recognizedTextElements.remove(intersectedText)
11:    else
12:        result.addNoLocationMatch(expextedText)
13:    end if
14:    index + +
15: end while
```

Now that we have created the relations between discovered text elements and the expected text elements, it was time to try to match the individual content of the text elements. Here, we used the expected text again as our baseline. The algorithm tried to match each individual character from the expected text, with a character from the linked OCR result. We started by sorting the linked OCR results based on their location. We accomplished this by creating a virtual grid for the OCR result by starting with the coordinates (0,0). Algorithm 6 shows the pseudocode, which creates this sorted grid of OCR results. The grid is created by first calculating the average line height and then grouping the OCR results based on their Y coordinate. Next, we sorted the groups on their X coordinate. This grid enabled us to iterate consistently over the OCR results. We needed this for content that consisted out of multiple OCR results. This was typically the case for content that consisted out of multiple words, E.g. "Save as". With this grid in place, we had installed a safety barrier that prevented us from making incorrect matches. Figure 4.11

shows the about dialog of the *Notepad* application. For the text element at the bottom of this figure, we queried the location and size of the widget. The widget was located at coordinate (147,411) with a width of 540 pixels and a height of 94 pixels. The expected text was *The Windows 10 Pro operating system and its user interface are protected by trademark and other pending or existing intellectual property rights in the United States and other countries/regions.* The list 4.1 below contains the outcome of our sorting algorithm for OCR results, which have been linked in the previous step to the expected text element. As we can see, the algorithm has calculated that three lines were required. Once the OCR results were grouped per line, we sorted them based on their X-coordinate so that we could match the individual characters.

Algorithm 6 Sorting the OCR results

```
1: calculateLineHeight()                          ▷ Based on average height of OCR results.
2: groupOCRResultsOnY_Axis()                      ▷ Each group represents a line.
3: while index! = groupedOcrResults.size() do     ▷ For each line:
4:     line = groupedOcrResults[index]
5:     sortOCRResultsOnX_Axis(line)               ▷ Sorted by the X-axis location of the OCR result.
6:     index++
7: end while
```

Figure 4.11: The about screen of Notepad.

- OCR results for calculate line at Y-coordinate 421.
  - text='The' at coordinate (146,413) width:27 and height:13
  - text='Windows' at coordinate (178,413) width:64 and height:13
  - text='10' at coordinate (248,414) width:17 and height:12
  - text='Pro' at coordinate (271,414) width:23 and height:12
  - text='operating' at coordinate (299,414) width:67 and height:15
  - text='system' at coordinate (371,414) width:48 and height:15
  - text='and' at coordinate (425,413) width:26 and height:13
  - text='its' at coordinate (457,414) width:15 and height:12
  - text='user' at coordinate (478,417) width:29 and height:9
  - text='interface' at coordinate (513,413) width:59 and height:13
  - text='are' at coordinate (577,417) width:22 and height:9
  - text='protected' at coordinate (605,413) width:65 and height:16

- OCR results for calculate line at Y-coordinate 439.
  - text='by' at coordinate (147,432) width:16 and height:16
  - text='trademark' at coordinate (168,432) width:73 and height:13
  - text='and' at coordinate (245,432) width:26 and height:13
  - text='other' at coordinate (276,432) width:37 and height:13
  - text='pending' at coordinate (319,432) width:56 and height:16
  - text='or' at coordinate (380,436) width:15 and height:9
  - text='existing' at coordinate (400,433) width:54 and height:15
  - text='intellectual' at coordinate (460,432) width:73 and height:13
  - text='property' at coordinate (540,433) width:59 and height:15
  - text='rights' at coordinate (605,432) width:39 and height:16
  - text='in' at coordinate (650,433) width:11 and height:12
- OCR results for calculate line at Y-coordinate 459.
  - text='the' at coordinate (146,451) width:22 and height:13
  - text='United' at coordinate (174,451) width:45 and height:13
  - text='States' at coordinate (224,452) width:42 and height:12
  - text='and' at coordinate (271,451) width:26 and height:13
  - text='other' at coordinate (302,451) width:37 and height:13
  - text='countries/regions.' at coordinate (344,451) width:126 and height:16

List 4.1: The sorted OCR results.

We started with the first character from the expected text. We iterated over the grid and tried to find a match. Once found, we marked the character from the expected text as matched and stored the index of the matching character from the grid. We used this index as our new starting point for the next character. This prevented us from making incorrect matches with characters located earlier in the grid. In case that the expected character and the discovered character in the grid were not matching, we extended the comparison by checking if there could be a potential casing mismatch. If this was indeed the case, we marked the expected character as matched with the note that it had a case mismatch. This could be caused by an OCR detection error.

During the implementation, we noticed that the expected text sometimes contained whitespace characters. Whitespace characters are by definition not detectable by our setup because we configured the OCR engine to detect words instead of sentences. In order to solve this problem, we mark all white space characters as whitespace corrected.

Once we have processed all the expected characters, we have our final matching result. For each expected character, we have set one of the following matching results.

- Matched.
- Case mismatch.
- Whitespace corrected.

- Not matched.

The sum of the results for all the individual characters will contribute to whether this could be a potential TPF. In case that all the characters are matched correctly, a TPF is obviously not detected. However, if there are "*Not matched*" results included, it could potentially be a TPF. Further manual inspection is required to determine whether our implementation has discovered a TPF. It is up to the reporting component to make a judgement and set the verdict based on the matcher results.

4.6 Reporting

Now that we have collected the outcome of the matcher, we had to present the result towards the user. We have accomplished this by incorporating the result into the reporting mechanism of TESTAR. For each sequence, TESTAR creates a dedicated HTML report. This report contains an overview of the executed test run. It starts with the initial state, followed by this sequence for the number of specified actions:

- An overview of all the available actions.
- The execution of the selected action.
- The new state of the SUT after completing the execution of the selected action.

And as last, the final state of the SUT together with the overall verdict of the test run is added to the report.

4.6.1 HTML report

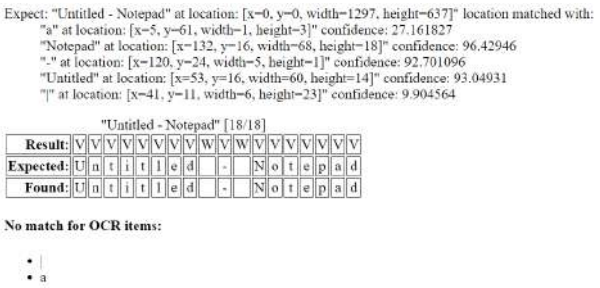
Because of the fine grained outcome of the matching algorithm, we extended the current HTML report by adding the matching result from two different perspectives. The first one provides a global overview of the matching result by showing an annotated screenshot of the analyzed input. Figure 4.12 shows the annotated screenshot. And the second one provides a detailed overview of the matching result for each expected text element. For the global overview, we annotate the screenshot that we have fed to the OCR engine with rectangles around the expected text. Because we do not want to change the original screenshot, we create a copy of the original screenshot. We will decorate each expected text element. Depending on the number of matched characters, we select a color;

- For 100% matched we use a green-coloured border.
- For a match result between the configured threshold and 100% we use a yellow-coloured border.
- For a match result below the configured threshold, we use an orange-coloured border.
- For the expected text elements that do not have a corresponding location match, we use a red-coloured border.

Figure 4.12: Overview of matched result.

In case that the matching algorithm could not find an OCR result that intersects with expected text location, we create a list that contains the expected text and the location of the element.

For the detailed view, we create for each expected text element an overview of the matched OCR results based on the location and a table with the matched result. The header row contains the result of the match. The next row contains the expected text, and each column represents a single character. The last row holds the matched character detected by the OCR engine. When applicable, we list the unmatched OCR characters below the table which were found within the location of the expected text. Figure 4.13 shows the detailed view of a match result.


Expect: "Untitled - Notepad" at location: [x=0, y=0, width=1297, height=637]" location matched with:
"a" at location: [x=5, y=61, width=1, height=3]" confidence: 27.161827
"Notepad" at location: [x=132, y=16, width=68, height=18]" confidence: 96.42946
"-" at location: [x=120, y=24, width=5, height=1]" confidence: 92.701096
"Untitled" at location: [x=53, y=16, width=60, height=14]" confidence: 93.04931
"|" at location: [x=41, y=11, width=6, height=23]" confidence: 9.904564

"Untitled - Notepad" [18/18]

Result:	V	V	V	V	V	V	V	V	W	V	W	V	V	V	V	V	V	V
Expected:	U	n	t	i	t	l	e	d		-		N	o	t	e	p	a	d
Found:	U	n	t	i	t	l	e	d		-		N	o	t	e	p	a	d

No match for OCR items:

- |
- a

Figure 4.13: Detailed report view of matched result.

4.6.2 Verdict

When the visual validation module is enabled during a test run, we also create a verdict based on the outcome of the visual validation result. We will incorporate the verdict of the visual validation in the final verdict of the test run. For our visual validation verdict, we define four different severity levels that are related to the expected text elements. The overall verdict severity levels are ordered from 0, indicating everything is all right till 1, indicating that TESTAR has encountered a serious problem while executing the test run.

For the visual validation module, we have defined the following severity levels:

1. 0.0 when the expected text has been matched for 100%, in this case we use "*Verdict.OK*".
2. 0.1 for a match result between the configured threshold and %100.
3. 0.15 for a match result below the configured threshold.
4. 0.16 for expected text elements without an OCR result.

In case that the verdict is not "*Verdict.OK*", we include the expected text in the message of the verdict to identify which element is responsible for the severity level. The visual validation verdict will be calculated based on the matched percentage. We store the verdict with the highest severity level while processing the outcome of the matcher algorithm. In case that we have multiple results that have a matched percentage below the threshold, we store the result with the lowest percentage of matched characters.

Chapter 5

Exploratory experiment

In this chapter, we will perform an exploratory experiment to validate that our implementation is able to detect TPFs. We will do this by testing the implementation against a modified SUT and defining a metrics for measuring the performance of our implementation. Modifications will be made to an example application which we will use as SUT. We will introduce TPFs which need to be identified by our implementation before we can conclude that our implementation works according to the design. The outcome of the exploratory experiment allows us to answer some research questions as well in this chapter.

5.1 Introducing the hypothesis

In chapter 4, we have designed and implemented a solution which is devoted to detect the presence of TPFs within the SUT. We will now focus on validating the implemented solution. The outcome of the validation allows us to answer research question 3:

How can the implemented solution, for detecting automatically whether the text of GUI components is entirely visible, be validated?

In order to validate the implementation, we define the following hypothesis:

> *TESTAR is capable of detecting TPFs by validating whether the expected text is visualised entirely.*

With the following exploratory experiment, we will verify our hypothesis. The validation process needs to look into whether our approach is valuable for the tester and being capable of detecting TPFs. To this end, we will run experiments where we use our implementation. We will start with a controlled environment experiment.

5.2 Controlled environment experiment

5.2.1 Procedure for executing the validation

During the validation process, we will start with composing a dedicated TESTAR configuration for each new SUT. This TESTAR configuration is tailormade for the SUT so that our implementation only processes expected text elements which are actually visible. The configuration consists of the following files:

- The `test.settings` file.
- The `ExtendedSettings.xml` file.
- Depending on the type of SUT:
  - `Protocol_desktop_generic.java` file
  - `Protocol_webdriver_generic.java`

Once created, we will run TESTAR with our implementation enabled. When TESTAR has finished, we will manually inspect every processed screenshot by our implementation and compare that with the expected text and the calculated matching result. During this process, we will perform the measurements as described in section 5.2.2.

For our controlled environment experiment we will run TESTAR with our visual validation module enabled while we use a SUT with existing TPFs. The purpose of the controlled experiment is to create a baseline and validate that the implementation can detect TPFs. We will start with an exploration phase by running TESTAR in combination with the SUT. During this phase, we will identify the widgets that our implementation should not process. Once we have composed the configuration for the controlled experiment, our implementation will only process the right expected elements. With this configuration, we will do a final test run. We will inspect and discuss the outcome of this test run and perform the measurements as described in section 5.2.2. For each executed action, we will validate whether our implementation composed the right result.

5.2.2 Metrics

To measure the capability of finding TPFs, we will look into the following metrics:

- Measure how many expected text elements our implementation receives for each particular state. This number will contribute to creating the context when we examine our filtering mechanism and the successfully matched text elements.
- Measure how many expected text elements our implementation expects, for each state, how many expected elements are left after the filter mechanism has been applied. This number will contribute to creating the context when we examine our filtering mechanism and the successfully matched text elements.
- Measure the number of expected widgets which we need to exclude despite that they have visible text. This will tell us how good the filtering mechanism is. The less valid widgets

we need to filter out, the better our implementation can check whether all expected text elements are visible.

- Measure the number of widgets which were not detected by the OCR engine because another widget covered them because of an executed action. This tells us how good the input is when querying the expected text. A higher number shows we can not properly identify when a widget is blocked on purpose. We will measure this by manually inspecting the matching result and identify whether the expected text element is blocked.

- Measure the number of widgets which have not been identified by the OCR engine while they are not excluded and visible. This will tell us how good the OCR engine can recognize expected text elements. A high number indicates that the OCR engine has detected more text elements. The more text elements have been detected, the more expected elements we can try to match. A low number shows that the OCR engine cannot detect the expected text elements with the used configuration.

- Measure the number of actual TPFs. This comprises expected widgets, which have been partially matched by the detected OCR results, while the expected text element is not entirely visible. The OCR engine identified the visible text correctly and our implementation has detected a TPF.

- Measure the number of widgets which have been detected by the OCR engine and have entirely been matched with the expected text. A high number indicates that our implementation was capable of identifying and matching the expected text. These expected text elements have been identified and are marked as completely visible. We will do this by manually inspecting the expected text element and verify what the OCR engine has identified.

- Measure the number of expected widgets which have been partially matched because of the detected OCR result. The expected text element is entirely visible. However, the OCR engine did not recognise the entire text correctly. A high number indicates that the OCR engine is not capable of detecting everything using the current configuration.

- Measure the number of expected widgets which have been partially matched because of our matching algorithm. The expected text element is entirely visible and the OCR engine did recognise the entire text correctly. However our matching algorithm was not able to match all the characters correctly. A high number indicates that our matching algorithm needs to be improved.

5.2.3 Defining and setting-up the controlled experiment

For the controlled experiment, we need a SUT which we can modify so we can inject TPFs. Therefore, we have identified the following requirements:

- The application needs to be cross platform compatible, so that test can be reproduced on different systems.

- The application needs to contain a GUI, so that we can inject and identify TPFs.

- For the controlled experiment, we need a SUT which we can modify so we can inject TPFs. This SUT will be developed by the researcher for the sake of this experiment and hence will be in C++ because that is the language he is most familiar with.

As the starting point of the SUT, we have selected the `Application example` from Qt[1]. `Application example` is a simple notepad application. This application meets our requirements and is suitable to be used as a starting point for the SUT in our controlled experiment. The primary goal is to validate that TPFs can be detected. Therefore we need to inject them into the selected application. We have chosen to inject IPFs.

A short overview of the changes we have made to the application[2]:

- We have added multi language support to the application. This makes it possible to inject an IPF for a specific language.
- We have added the possibility to start the application with a specified language. This makes it possible for TESTAR to run the SUT with a specific language and try to identify the injected IPF.
- We have adjusted the dimensions for a couple of components. With these adjustments we introduce the IPFs. In the next paragraph we will elaborate on the adjustments we have made.

As mentioned in the introduction 1, TPFs do not visualize the expected text entirely. We can categorize incorrect visualized text into three groups. Starting with a missing punctuation. This could be a missing dot, for example. In this case, the sentence would still be readable. However, it is not grammatically correct. The second group covers a missing character. In most cases, the text would still be understandable. However, it could become more difficult for the user to interpret the sentence. We devote the last group to missing one or more words. When a user needs to read this, it will be hard to understand what the sentence is about. During the proof of concept 3.2, we have seen that we can obtain the expected text and that we can validate the expected text with the help of the OCR engine. However, we have not validated that our implementation can identify TPFs. In order to prove that our implementation can detect TPFs, we need to prove that the implementation is capable of detecting them. Since the initial version of the `Application example` did not contain any TPFs, we have added them. We have embedded three TPFs within our reference application. Each implemented TPF reflects a group. We start with a missing punctuation and will end with an entire sentence.

1. A TPF containing a punctuation which is not visible, see figure 5.1.
2. A TPF containing cutoff characters, see figure 5.2.
3. A TPF containing a sentence which is not visible, see figure 5.3.

For the first TPF, we set a fixed size. The size is obtained by consulting the font metrics with the original text minus the last character. The original text is `Active...`, the presented text will only show 2 dots.

We created the second TPF by forcing a fixed width for the `Help` menu. The original text is `About Qt`. Depending on the selected language, the second entry shows a small part of the second word, see figure 5.2, or only the first word `About`.

The third TPF is realized by setting a fixed height for the dialog. The original text is `The` **Application** `example demonstrates how to write modern GUI applications using Qt, with a`

[1] https://doc.qt.io/qt-5/qtwidgets-mainwindows-application-example.html
[2] https://github.com/TMenting/testar-vv-reference/releases/tag/v1.1

Figure 5.1: A punctuation presentation failure.

Figure 5.2: A cutoff character presentation failure.

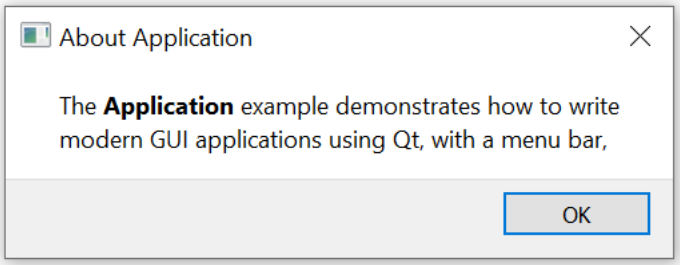


Figure 5.3: A missing sentence presentation failure.

`menu bar, toolbars, and a status bar.` This fixed height limits the visible part of the original text to two lines, see figure 5.3.

5.2.4 Composing the configuration

Now that the reference application for the controlled experiment is ready to be used, we needed to update the TESTAR configuration used by our implementation. The default TESTAR configuration has been composed to filter out the widgets which we do not want to be processed by our implementation. In general this is because these widgets do not contain any visual text while the querying the widgets returns a text. This is often a description of the widgets function, e.g. *scrollbar*. To create this new configuration for our reference application we have chosen for an iterative approach which started with an empty configuration file:

- Running our reference application as SUT with our implementation enabled while TESTAR used the configuration file.
- Once the run was completed we inspected the outcome and identified which widgets needed to be excluded from our analysis. Identifying which widgets need to be excluded was accomplished by inspecting the annotated screenshots. Widgets which did not contain any visible text were identified as false positives.
- The identified widgets were added to the configuration file.

This process was repeated four times until the outcome of our implementation only contained widgets, which actually contained visible text. Listing 5.1 shows partially the composed configuration file containing the excluded widgets and which criteria is used to exclude the widget.

The entire configuration can be found in the appendix A.2. The enumeration overview below corresponds with the comments in listing 5.1. For example the first item in the enumeration is related to line 3 in the listing. All the widgets listed in listing 5.1 did not contain any visible text, except for the last item. This was added because of a problem with the screenshot. At the end of this section will elaborate more on this problem.

1. All the buttons in the title bar section. E.g. `Maximize`.
2. All the toolbar buttons. E.g. `New`.
3. The toolbar sections. E.g. `Edit`. (Only visible in Spy mode.)
4. The upper left icon in the title of the dialog screens.
5. The upper left icon in the title of main application.
6. The close button of the dialogs.
7. The title bar of the main application.

Listing 5.1: Configuration settings

```
1  <widgetTextConfiguration>
2  <loggingEnabled>false</loggingEnabled>
3  <widget> <!-- #1 -->
4      <role>UIAButton</role>
5      <tag/>
6      <ignore>true</ignore>
7      <ancestor>::UIATitleBar::UIAWindow::Process</ancestor>
8  </widget>
9  <widget> <!-- #2 -->
10     <role>UIAButton</role>
11     <tag/>
12     <ignore>true</ignore>
13     <ancestor>::UIAToolBar::UIAWindow::Process</ancestor>
14 </widget>
15 <widget> <!-- #3 -->
16     <role>UIAToolBar</role>
17     <tag/>
18     <ignore>true</ignore>
19     <ancestor/>
20 </widget>
21 <widget> <!-- #4 -->
22     <role>UIAMenuItem</role>
23     <tag/>
24     <ignore>true</ignore>
25     <ancestor>::UIAMenuBar::UIATitleBar::UIAWindow::UIAWindow::Process</
           ancestor>
26 </widget>
27 <widget> <!-- #5 -->
28     <role>UIAMenuItem</role>
29     <tag/>
30     <ignore>true</ignore>
31     <ancestor>::UIAMenuBar::UIATitleBar::UIAWindow::Process</ancestor>
32 </widget>
33 <widget> <!-- #6 -->
34     <role>UIAButton</role>
35     <tag/>
36     <ignore>true</ignore>
```

```
37      <ancestor>::UIATitleBar::UIAWindow::UIAWindow::Process</ancestor>
38  </widget>
39  <widget> <!-- #7 -->
40      <role>UIAWindow</role>
41      <tag/>
42      <ignore>true</ignore>
43      <ancestor>::Process</ancestor>
44  </widget>
45  </widgetTextConfiguration>
```

During this iterative process, we noticed that TESTAR entered dialogs that were steering away from the main functionality of the reference application. We identified the save and open action as triggers for these flows. Once these dialogs were opened, the chances were small that we would return to the main screen of the reference application. Therefore, we decided to extend the action exclude filter mechanism. We extended the filter with the following expression to prevent TESTAR from selecting these actions: `Open.*|Opslaan.*| Afsluiten`

While composing the final TESTAR configuration for the reference application as SUT, we made the following observations.

- The screenshots of the reference application were cut off. The title bar and the related buttons, minimize, maximize and close, of the main window were not included in the captured surface, as we can see in figure A.1. Because of this, it was not possible for the OCR engine to detect the filename which is presented in the title bar. We have decided to mark this as a bug within TESTAR when using Qt applications as test targets. We have added the title bar to our ignore filter in the configuration file of the visual validation as a workaround for this bug. This prevents that our implementation attempts to validate the filename shown in the title bar.
- We can prevent having duplicates in the exclude configuration by supporting a wildcard pattern for the ancestors tag in the configuration file. For example, listing 5.1 contains three entries to ignore a "*UIAButton*", entry one, two and six. The only difference is their ancestor path. With a wildcard pattern we could reduce the ancestor path to for example: `UIAT.*Bar.*:UIAWindow::Process`. This would make it obsolete to exclude the close button for each "*UIAWindow*".
- While analyzing the generated reports by TESTAR, we noticed that there was no distinction between expected text elements that did not have a location match with OCR results and OCR results that were not expected. Therefore, we have introduced purple as additional border color. We use this color for OCR elements that were not expected. Figure A.1 reflects this scenario, the icons of toolbar buttons are recognized by the OCR engine but were not expected.

5.3 Results of the evaluation with the controlled experiment

We will now examine the report from TESTAR with the `Application example` as the SUT. Because of the limited amount of available actions within the SUT, we have chosen to run a single sequence with ten actions. As stated previously, we have excluded the actions that will trigger the `open` and `save` dialogs. A detailed review for each step of the obtained results has been added to the appendix A.3.

A short summary of our findings while using the visual validation implementation with the reference application:

- The three TPFs have been identified correctly by our implementation. This shows that our implementation is capable of detecting TPFs.
- The expected text element `Actief..` has multiple times incorrectly been marked as a 100%. This is because of an incorrect OCR detection.
- Optimizing the OCR engine is required to eliminate incorrect outcomes from our implementation. E.g. the OCR engine did not detect properly the expected text `OK` of the button in the dialog.
- Some potential improvements:
  - The implementation might benefit from taking the confidence levels into account. However, we have also seen that sometimes, even with a low confidence level, the recognized text was correct. Embedding the confidence level while matching the character has to be done carefully to prevent incorrect results.
  - The visual validation implementation can be extended by incorporating a spellchecker.

Screenshot No	Expected elements	Expected elements filtered	Excluded despite visible text	Covered by action	Not detected by OCR	TPF	Entire matches	Partially matched	
								OCR	Algorithm
1	27	6	1[1]	0	0	1	5	0	0
2	1	1	0	0	0	0	1	0	0
3	33	7	1[1]	0	0	1	6[2]	0	0
4	1	1	0	0	0	0	1	0	0
5	34	8	1[1]	0	1[1]	1	5[2]	1	0
6	1	1	0	0	1	0	0	0	0
7	27	5	1[1]	0	0	0	5[2]	0	0
8	1	0	0	0	0	0	0	0	0
9	27	6	1[1]	0	0	1	5	0	0
10	1	1	0	0	0	0	1	0	0
11	33	7	1[1]	0	0	1	6[2]	0	0
12	1	1	0	0	0	0	1	0	0
13	34	8	1[1]	0	1[1]	1	5[2]	1	0
14	1	1	0	0	1	0	0	0	0
15	27	5	1[1]	0	0	0	5[2]	0	0
16	1	1	0	0	0	0	1	0	0
17	35	11	1[1]	0	0	0	8	0	0
18	1	1	0	0	0	0	1	0	0
19	27	6	1[1]	0	0	0	6[2]	0	0
20	1	1	0	0	0	0	1	0	0
21	35	9	1[1]	1	0	0	8[2]	0	0

Table 5.1: Measurements reference application

[1] The window title is excluded due to screenshot problems.
[2] The OCR engine detected three dots while only two were visible.

The result as shown in table 5.1 shows that our implementation is capable of matching the expected text with the detected text elements. For most of the screenshots, the number listed in the `Expected elements filtered` column are matched by summing up the columns; TPF, *Entire matches* and *Partially matches*. In addition, our implementation also detected the injected TPFs. The OCR engine detected most of the expected elements correctly. This high detection rate contributed to a high number of `Entire matches` text elements. Overall, there are no major concerns when we look at the outcome of the matching result for the reference application. These results confirm our hypothesis, defined in section 5.1. TESTAR can indeed detect TPFs with our implemented solution by validating whether the expected test is visualised entirely.

Now that we have validated that our implementation is capable of detecting TPFs, we will test the implementation against six different target applications to check whether we can find real TPFs. The collection will comprise three desktop applications and three websites. For these test runs, we will not discuss each individual step, but only the results that are not as expected and perform the measurements as defined in section 5.2.2. Once we have examined all the results, we will provide recommendations for improving the implementation.

5.4 Research question answers

Now that we have implemented our design of our visual validation module, we can answer some of the research questions. We start by providing an answer to our first research question 1.

How can OCR help to automatically validate whether the text is entirely visible?

We have seen from the first test results in figure 4.4 during the implementation phase that the OCR engine is capable of recognizing a high percentage of the expected text. This enables us to create an algorithm that can determine which text elements are presented correctly based on the recognized text elements by the OCR engine. By providing the captured screenshot from the SUT, the OCR engine can return the recognized text and the corresponding location. By providing a screenshot to the OCR engine, we have a validation system that is decoupled from the part that contains the knowledge about the SUT and the expected text. The screenshot does not contain any other information than the captured screen of the SUT. Therefore, we can consider the obtained results from the OCR engine to be objective and independent. We have confirmed this by the first results of the webdriver and desktop protocol test runs. The OCR engine returned more results than expected. For example, the recognized toolbar buttons and the *Open Universiteit*, which was located inside an image.

We will now answer research question 2a

How to combine the OCR results when the expected text consists out of multiple words?

In order to solve the problem that the output of the OCR engine consists of individual words, we have designed and implemented a solution based on the location of text. By using the location of the expected text as reference location, we iterate over the OCR results and match the results that intersect with the reference location. To prevent that we match OCR results with the incorrect expected text location, we sort the expected text results. By sorting the surface area of the expected text from small to large, and processing them in the same order, we prevent that an OCR result is being assigned to a larger surface area, when it should be linked to a smaller one. In the scenario of the expected text ' Ln 1, Col 1' in 4.5 which has a

location of [x=777,y=807,width=208,height=34], we will link the OCR results that are located within the range of the X axis 777 - 985 and Y axis: 807 - 841. Looking at the OCR result 4.1, we successfully link result:

- 15: 'Ln' [x=790,y=818,width=16,height=13]
- 16: '1,' [x=814,y=818,width=11,height=15]
- 17: 'Col' [x=831,y=817,width=25,height=14]
- 18: '1' [x=864,y=818,width=5,height=13]

An additional benefit that we have gained by using the expected text as a starting point, is that after the location matching is finished, the remaining OCR results can be classified as not expected and can therefor be left out of scope for the remaining part of the matcher algorithm. As an example, these can be the toolbar buttons from the first OCR test result of the desktop protocol or the *Open Universiteit* text located inside an image.

Another problem that we have addressed and solved allows us to answer 2b.

How can we filter out the false positives of widgets that do not have a text that needs to be visualized?

To be able to filter out the false positives, we first need to complete an initial run of the SUT while our implementation is enabled. Once this run has completed, we need to examine the results and look for the items that have not been matched successfully. Now that we have identified these results, we need to start the SUT again, but this time we need to run TESTAR in the "*SPY*" mode instead. In this mode we can lookup the widgets of interest and validate whether we need to filter them out. If this is the case, we have two options. Either we exclude the widget based on its type, in this case we will exclude all widgets of this type. Or we exclude this particular widget based on its location. We accomplish this by looking up the ancestor path. Once we have collected the required information about the false positives, we can compose a dedicated entry (see section 4.4) in our new settings framework. There is one special scenario in which we automatically filter out false positives. This only applies for when the SUT is being used in combination with the "*webdriver protocol*". When the expected text is positioned off screen, e.g. the page is too long and contains a scrollbar. We check for each widget if the property "*WebIsFullOnScreen*" is enabled. If this is not the case we filter out that particular widget.

With the implementation finalized we can also now give an answer to 2c.

How to match the results from the OCR engine with the information that is extracted from the accessibility API?

For matching the OCR results with the expected text, we start each round with 2 sources of input: the expected text and the matched OCR results based on their matched location. Because the expected text can consist of multiple sentences, we had to structure the OCR results. We have accomplished this by creating a grid for the OCR results based on their location. We first group the OCR results based on their Y location, this is done by calculating the average text height and then grouping them based on each calculated line. Next, we sort each group based on their X location. This gives us a grid that is sorted in the left to right and top to bottom reading order. We now start the comparison by iterating over the expected

text, and for each reference character, we try to find a matching character in the grid. Once found, we mark the location and use that as our new starting point for the next character. This prevents us from creating matches with characters matched earlier in the matching process. We mark whitespace characters as matched by default. This is because of our configuration of the OCR engine. By returning individual words, whitespaces are not included. In addition, the OCR engine can not detect leading and trailing whitespaces. If the reference character can not be matched with the current index of the grid, we lower the bar by dropping the case sensitivity. If this results in a match, we consider the reference character to be matched and make a note that we dropped case sensitivity. All reference characters that could not be matched with a character from the grid will remain marked as unmatched.

At last, we can answer the final research question 2 related to the implementation of the visual validation module:

How can OCR be integrated into TESTAR in order to validate the visibility of the text for the application under test?

To be able to validate the visibility of the text for an application under test, it is mandatory to know what the expected text and the presented text are. In section 4.1.2, we present an overview of how the visual validation module should be incorporated into the current codebase. We have identified four steps that the visual validation module consists of;

1. Extracting presented text.
2. Collecting the expected text.
3. Matching the expected text with the presented text.
4. Composing a verdict based on the outcome of matching.

By using an OCR engine, we are able to detect and collect the presented text from the application under test. In order to obtain the presented text, we need to feed the OCR engine with the screenshots that are captured by TESTAR. The created design shows us at which points we had to intercept the process flow of TESTAR and start the visual validation module. By starting the visual validation module, we feed the screenshot to the OCR engine. By querying TESTAR, we can collect the expected text for the visual UI components, and with the output of the OCR engine, we have our counterpart for the validation process. We have implemented a matching algorithm that takes care of the validation process. The algorithm can determine whether the expected text is visible. We accomplish this by identifying if there is a mismatch between the OCR detected text and the expected text. We embedded the outcome of the validation process into the reporting mechanism of TESTAR. The reporting mechanism composes an HTML report that presents the result of the visual validation in a user friendly overview.

Chapter 6

Validating the implementation

In this chapter, we will test our implementation with six SUTs. We will compose for each SUT a dedicated configuration and elaborate on the decision we have made. Once the configuration has been created, we will use the configuration combined with our implemented visual validation module to test the SUT. The outcome will be discussed and we will apply the metrics, as have defined in section 5.2.2.

6.1 Experiment with six different target applications

For selecting the desktop applications we used the following criteria.

- The desktop applications need to support the Windows environment. This because TESTAR has good support for Windows environments. Using a proven environment reduces the changes of spending resources on getting the test up and running.
- The applications need to be easily accessible so that someone else can repeat the test.
- The applications need to be familiar by the majority of people. This makes it easier to interpret the results without the need to get familiar with the SUT.

We have selected the following desktop application:

1. Paint
2. Microsoft Word
3. Windows Media Player

All three applications support the Windows environment. Paint and Windows Media Player are pre-installed in a windows environment and therefore well known. Microsoft Word is one of the major players when it comes to word processing software. Therefore, it is assumable that majority of the people are familiar with this software package and that it is most likely that it is available in the environment.

We have selected the following websites:

1. `www.ou.nl`. Because TESTAR uses the website of the university as a showcase, we wanted to include this website to test our implementation.
2. `www.nytimes.com`. We have selected this website based on the content. The website consists mostly out of text and has a plain layout.
3. `www.apple.com`. We have selected this website because it supports a lot of different languages.

6.2 Paint

6.2.1 Composing the configuration.

While composing the configuration we encountered the following two problems:

The first problem we encountered was that depending on the available space the visualization for certain buttons changes. E.g. `Cut` and `Copy` show this behavior when resizing the window. In figure 6.1 only the icon is shown, while in figure 6.2 both the icon and text are shown of the button. We have tried to find a difference in the tag values while using the "*Spy mode*". Unfortunately the values were all identical between the two visualizations. This means that our implementation will mark these buttons as failures when the text is not visible. In addition, figure 6.3 shows "*UIAButtons*", e.g. `Pencil`, which will never show its expected text. However, if we decide to ignore the buttons based on their ancestor path we would also ignore the `Color` buttons as shown in figure 6.4. Therefore, we have decided to make the window width enough, so that the text for buttons like `Cut` are visible. And ignore the result for the six buttons in the `Tools` section. Based on this problem we conclude that with the current exclude pattern for ancestors it is not possible to ignore individual elements. The ancestor path is composed out of the elements types of the ancestors, by replacing this with the actual "*Path*" tag we can improve the filter mechanism.

Figure 6.1: Toolbar buttons with only an icon.

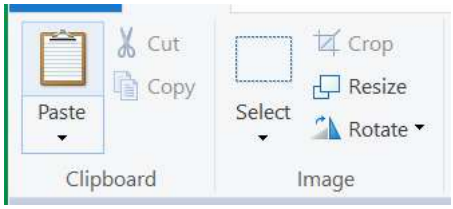


Figure 6.2: Toolbar buttons with both icon and text.

Figure 6.3: Toolbar buttons without text.

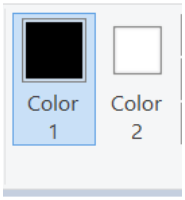


Figure 6.4: Toolbar buttons `Color 1` and `Color 2`.

The second problem was that we noticed that the expected text contained duplicates for the "*UIASplitButton*" elements. The problem with the "*UIASplitButton*" is that visually

it may seems like one button, composed out of text and image. Figure 6.1 contains two "*UIASplitButtons*", `Paste` and `Select`. However the "*Spy mode*" showed us that they are two individual elements. Unfortunately we can not identify which one is the icon and which is the text. Both elements contain the same expected text. Because each "*UIASplitButton*" is nested within a "*UIAGroup*" which had the same text as the "*UIASplitButton*" element, we decided to ignore the "*UIASplitButton*" widget based on its type. This decision however resulted in a side effect. The "*UIAGroups*" are nested within a larger "*UIAToolBar*". Figure 6.1 shows two of them, `Clipboard` and `Image`. The SUT contains a "*UIAToolBar*" with the expected text `Shapes`. However, that "*UIAToolBar*" also contains an "*UIAGroup*" with the same expected text. This "*UIAGroup*" however does not contain any visible text. It is only a box which contains the different kinds of shapes that can be drawn on the canvas. Because our implementation, the ancestor path exclusion is not specific enough. Therefore, we can not exclude this element from our results.

6.2.2 Inspecting the results.

We have chosen to perform one sequence of consisting out of eight actions. Because of the optimized configuration only interesting actions will be available to be selected. The application also does not contain many dialogs which will introduce new text elements. Therefore eight was considered to be enough for the test.

- While analyzing the outcome of the test run, in general we see that the OCR engine returned a lot of ghost characters. We have seen this before by our reference application. This is mainly caused by the icons and images. The presence of these ghost characters influenced the overall result of the OCR engine. A lot of expected text elements have not been detected by the OCR engine.
- As predicted the report contains duplicates for the expected text `Shapes`. The "*UIAToolBar*" is correctly matched with the OCR result. However the "*UIAGroup*" element could not be matched because there is no visible text.
- The expected text `531,202?px` and `1425 x 597?px`, located in the bottom left corner of figure 6.5 could not be matched completely. After examining the outcome we concluded that this is because of a special character `?` which is incorporated in the expected text. The character is visualized in the report as a question mark however this is not the original visualization. Further investigation is required to identify which character is expected. Our implementation needs to be extended to support ignoring expected characters. Based on the manual inspection we concluded that this character is not detectable for the OCR engine. Therefore we mark this result as correctly matched.
- For the expected text `Edit with Paint 3D` the OCR engine detected the following text; `9`, `Paint` and `3D`. Our matcher algorithm converted the uppercase D to a lowercase d and matched with the second character of the expected text. Because our implementation stores the index of the last successful match, there was no possibility that `Paint` and `3` could be matched since they had been passed to match the `D`. We can improve our implementation if we postpone the case matching logic until all expected characters have been processed.

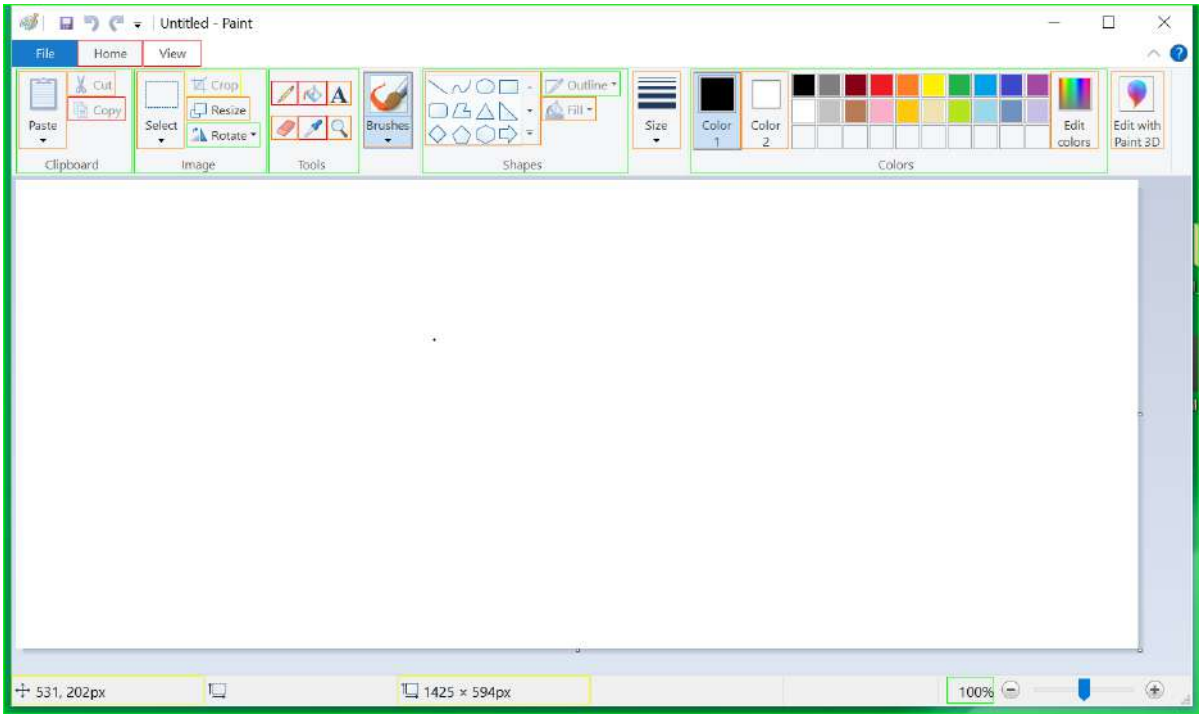


Figure 6.5: The initial state of the paint as SUT.

Table 6.1 shows an overview of the matching results for the initial state of the SUT. This overview is representative for the entire test run. The results are sorted by their matched percentage from high till low. The strike through style is used to indicate that the expected text has been match correctly. The characters with the normal text style have not been matched.

Expected text	Percentage	Comment
~~Untitled - Paint~~	100%	
~~Colors~~	100%	
~~Shapes~~	100%	
~~Image~~	100%	
~~Clipboard~~	100%	
~~Tools~~	100%	
~~Outline~~	100%	
~~Rotate~~	100%	
~~100%~~	100%	
~~531, 202~~?~~px~~	91%	Can be interpreted as 100% because of the special character
~~1425 x 594~~?~~px~~	85%	Can be interpreted as 100% because of the special character
~~Crop~~	75%	
Edit ~~colors~~	64%	
Cu~~t~~	33%	
~~Sha~~pes	33%	This was expected
Color ~~1~~	29%	
Color ~~2~~	29%	
~~S~~ize	25%	
Edit with ~~Paint 3D~~	22%	
Select	0%	
Paste	0%	
Brushes	0%	
Fill	0%	
Resize	0%	
Copy	0%	
View	0%	
Home	0%	
Eraser	0%	This was expected
Magnifier	0%	This was expected
Text	0%	This was expected
Pencil	0%	This was expected
Fill with color	0%	This was expected
Color picker	0%	This was expected

Table 6.1: The matching result of the initial state of Paint.

Figure 6.6 shows the state after the `Paste` button has been clicked. The expected text `Clipboard` could not be found. This is visualised by the red outline above the text `Paste from`. This is however expected because of the popup menu shown by the click action. For the following expected text items the matching result has significantly improved compared to the initial state.

- ~~Crop~~
- ~~Brushes~~
- ~~Size~~
- ~~Color~~ 1
- ~~Color~~ 2

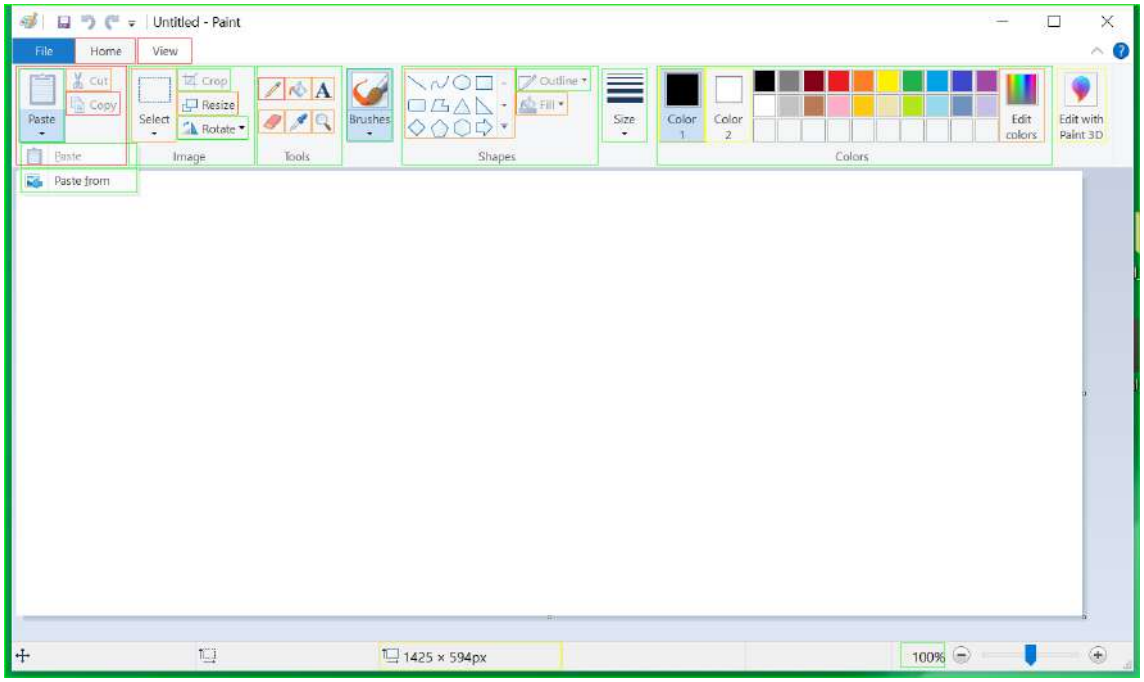


Figure 6.6: The state of the paint after clicking on `Paste`.

- ~~Edit color~~
- ~~Edit with paint 3~~D
- ~~Paste~~
- ~~Paste from~~

Despite of the relative small popup window there are no differences between the screenshot taken from the initial screenshot and after the first action. We could improve our matching algorithm if we can combine individual results across different states together. E.g. the `Crop` has successfully matched in figure 6.6 but only partially in figure 6.5, despite of the fact that they are visually identical.

During one of the actions TESTAR has clicked on the File tab. After this we see that the ribbon is partly covered and that all the expected items could not be found. Figure 6.7 shows this state. This is similar to the small popup in 6.6 which was blocking the expected text `Clipboard`. While these outcomes are valid, these can not be classified as TPFs. Our current implementation can not distinguish whether the expected element was temporarily blocked.

Figure 6.7 shows the `Recent Pictures` section. This section contains ten screenshot names captured by TESTAR. While investigating the results we have identified the following problems:

- The expected item `SCC10bysbb2002148710592.png` was found multiple times in the overview. For the second entry the following OCR results have been found `10592.png`, `SCC1Obysbb20021487` and `bo`. Because the OCR result identified the first `0` as an `O` the expected zero was matched with a zero found later on in the OCR result. Because of this correctly detected characters were skipped, while the first entry was detected correctly.
- The same has happened for the expected item `SCC1c4bq3w1fc1611254898.png`. The OCR result was `SCC1e4bq3w1fc1611254898,png`. The first `c` could not be found in the OCR

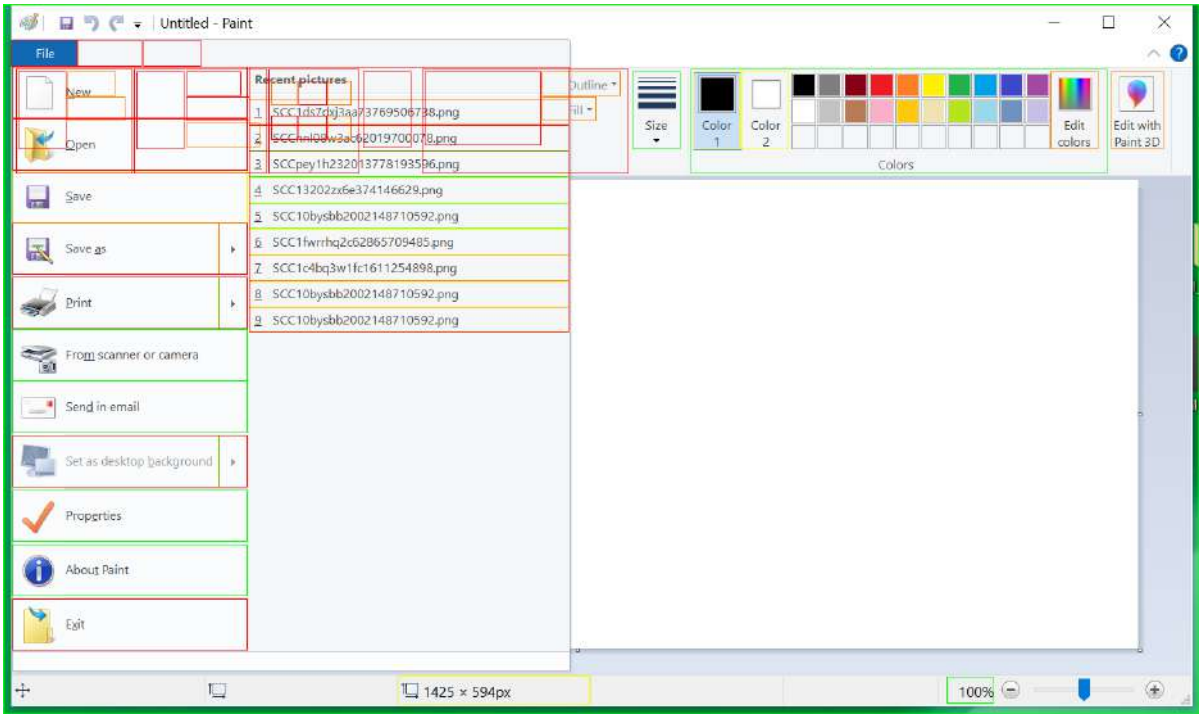


Figure 6.7: The open recent view of paint as SUT.

result and therefore it was matched with a `c` later on while the characters between were skipped.

- Because we match OCR results with expected elements based on their location some items in the recent overview got matched with elements that were located behind the overview. The OCR result `scCnnloewsees2019700078` for expected text element `SCCnnl08w3ac62019700078.png` was linked to `Magnifier`. The same goes for expected element `SCC1ds7dxj3aa73769506738.png`, the OCR result `8Â¢C1ds7692a73769506738.pn9` was linked to `Magnifier` as well.

After analysing the generated report we noticed that we had made a minor mistake in the configuration file. Instead of reassigning the right tag for "*UIAEdit*" to use the tag "*UIAValueValue*", as we did during the implementation phase, we have configured it to be ignored instead. However by inspecting the images in the report we can conclude that the OCR engine did not recognized them. Because if the OCR results could not be matched with an expected text element they will be highlighted with a purple outline. As there are no purple outlines visible we conclude that the "*UIAEdit*" text elements have a 0% match result.

Table 6.2 shows that the OCR engine did not perform as good as it did for the reference application as in table 5.1. The OCR engine failed to detect 100 text elements from a total of 284 expected text elements. In addition, we could only partially match 46 expected text elements because of the OCR result. 35% of the expected text elements were matched entirely by our algorithm and 22 expected text elements were only partially matched because of our matching implementation. The filtering mechanism returned 324 expected text element while, after manually inspecting the application, we counted 284 expected text elements. This means that our implementation should not process that 12% of the expected elements. These expected text elements negatively affect the outcome of the validation result. Finally, 14 expected text elements could not be detected by the OCR engine because the executed action hid them.

Screenshot No	Expected elements	Expected elements filtered (actual)	Excluded despite visible text	Covered by action	Not detected by OCR	TPF	Entire matches	Partially matched	
								OCR	Algorithm
1	100	33[1](26)	1	0	9	0	9	6	2
2	1	0	1	0	0	0	0	0	0
3	109	34[1](26)	1	1	5	0	14	5	2
4	1	0	1	0	0	0	0	0	0
5	140	60[1,2](33)	0	14	5	0	12	8	8[3]
6	1	0	0	0	0	0	0	0	0
7	100	32[1](25)	0	0	10	0	8	6	1
8	1	1	0	0	0	0	1	0	0
9	129	50[1,4](40)	0	1	19	0	10	8	2
10	1	0	1[5]	0	0	0	0	0	0
11	129	50[1,4](40)	0	1	22	0	11	5	2
12	1	1	0	0	0	0	1	0	0
13	129	50[1,4](40)	0	1	11	0	19	8	2
14	1	1	0	0	0	0	1	0	0
15	100	33[1](26)	0	0	9	0	9	6	2
16	1	0	0	0	0	0	0	0	0
17	100	32[1](25)	0	0	10	0	12	2	1

Table 6.2: Measurements Paint.

[1] Includes six "*UIABUttons*" in `Tools` section and one "*UIAGroup*" in `Shapes` section which do not contain text. We could not exclude this widgets properly without excluding widgets which do contain text.

[2] Incorrect filter mechanism for "*UIAGroup*" in the `File` tab affecting, `Save As`, `Print` and `Set as desktop background`. This leads to three times the expected text instead of only once.

[3] Includes filenames which could have been matched better when improving our algorithm. However a 100% match is not possible because some characters are still detected incorrectly by OCR engine.

[4] Edit dialog has a hidden button which was not excluded by our filter. Also some widgets in the dialog were not properly added to the filter.

[5] Accidentally included the "*UIEdit*" widget in the exclude filter. While we actually needed to specify the "*UIAValueValue*" tag instead.

6.3 Microsoft Word

6.3.1 Composing the configuration.

For running Microsoft Word as SUT, we use two additional arguments.

- `/q` This makes Microsoft Word startup without showing a splash screen.
- `/w` This makes Microsoft Word startup with a new blank document.

While executing some exploration runs to identify which elements we need to ignore when our visual validation module is enabled, we noticed two new problems.

- The first problem was that the generated directory which contains the reports did not

include the application name. We have solved this by first extracting the absolute path of the application. This prevents that the logic take the arguments into account.

- The second problem was that the screenshot sizes were not correct. After investigating the problem, we found out that there were four so called `MSO_BORDEREFFECT_WINDOW_CLASS` widgets without child widgets under the root element and one `OpusApp` widget with multiple child widgets. Therefore, we have changed the selection logic, which determines the dimensions of the SUT. Instead of always selecting the first element, we now select the first element which contains child elements.

Some widgets need to be excluded while they have visible text. This because of other widgets of the same type which do not contain visible text. One of these widgets is the "*UIAButton*". The majority of these buttons only contain icons and no text. However, there are three types which will be excluded while they should not. Figure 6.8 shows the ignored buttons.

- Find
- Replace
- Select

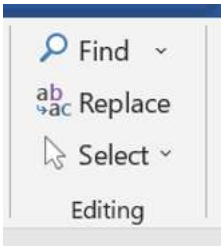


Figure 6.8: Buttons which will be ignored despite the visible text.

6.3.2 Inspecting the results.

We have configured TESTAR to execute ten actions within one sequence. Microsoft Word offers many different possibilities for text processing, presented by various GUI elements and dialogs. Because of this, executing ten actions is enough to test different states of the application. In addition, the large number of GUI elements of Microsoft Word makes it very difficult to create an encompassing filtering configuration.

- While inspecting the results from the TESTAR run with the Microsoft Word application, we noticed that the OCR engine failed to identify all the expected text elements. Especially for the initial state 6.9, the OCR engine failed to identify the tab headers: `Home`, `Insert`, `Design`, `Layout`, `References`, `Mailings`, `Review`, `View` and `Help`. However, for example, in screenshot 4 6.10 we see that some of the tab are identified correctly by the OCR engine. While analyzing the screenshot, we see that there are a lot of assets visible. These assets could lead to a lower number of identified elements by the OCR engine.

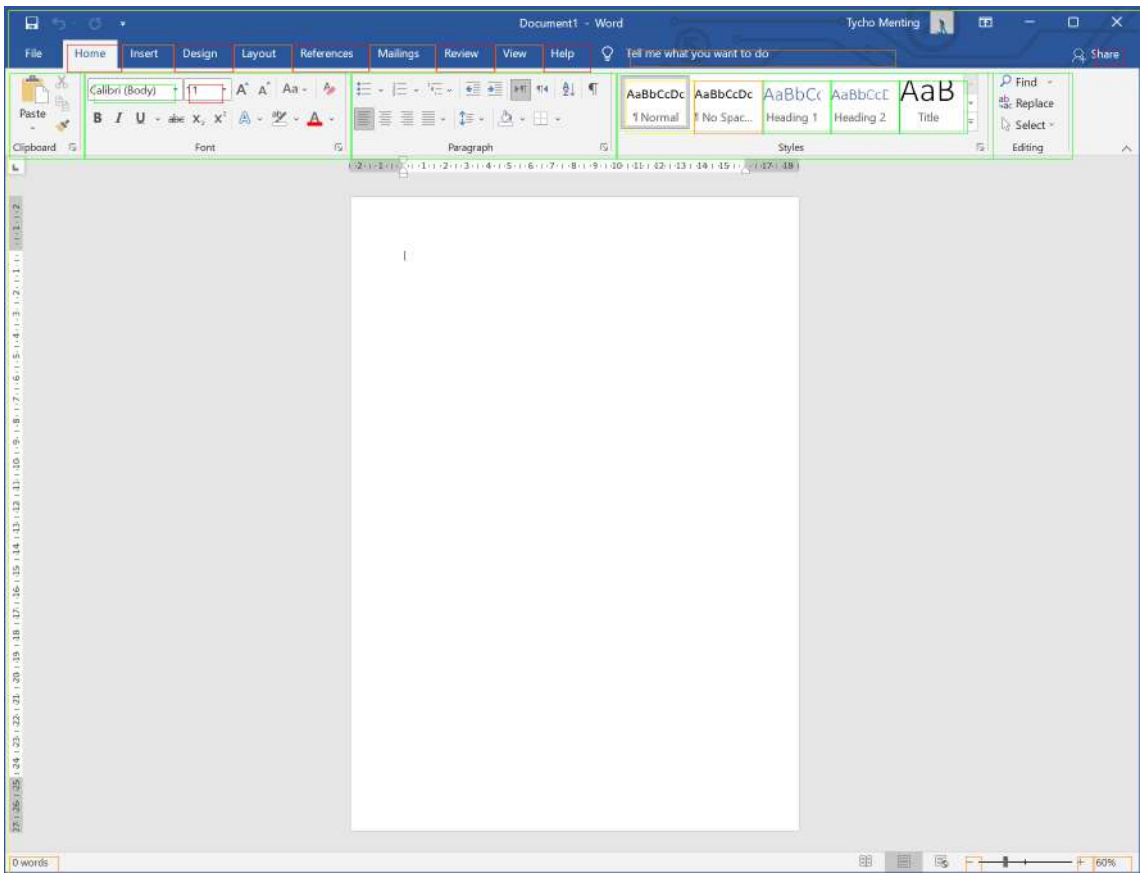


Figure 6.9: The initial state of MS Word.

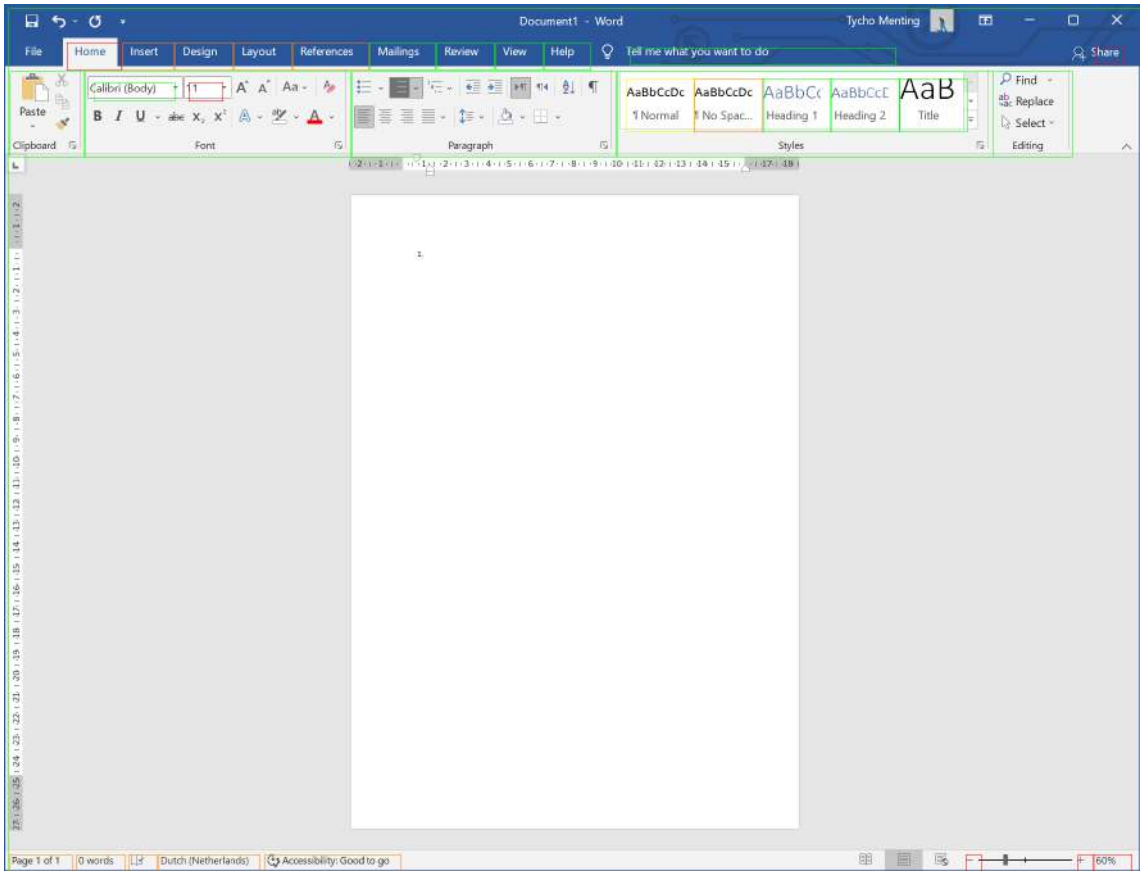


Figure 6.10: The fourth state of MS Word.

- Another point of interest for the initial state is the usage of special characters. In the upper section of the screenshot 6.9, we see the `Styles` group. The expected text element `¶ Norma` is positioned within this group. The OCR engine did not identify the special character ¶. Also, while the expected text consists out of `Norma`, the discovered text was `Normal`. The fact that the character `l` was not expected while it was visible is noteworthy.

- State four of Microsoft Word as SUT, shows a dialog as we can see in screenshot 6.11. The width of this dialog exceeds the width of the application. This resulted in an incomplete screenshot of the current state. Because of the incomplete screenshot, the OCR engine could not detect the entire expected text. The characters '`e`' and '`?`' were not visible and therefore not detected. Our matching implementation made a small mistake by matching the '`"`' character of the discovered text `*-1216109715"` with the first '`"`' of the expected part `"-1216109715"`. Therefore, it was not possible to match the number `-1216109715`. Besides this, the remainder of the expected text within the dialog was successfully detected and matched.

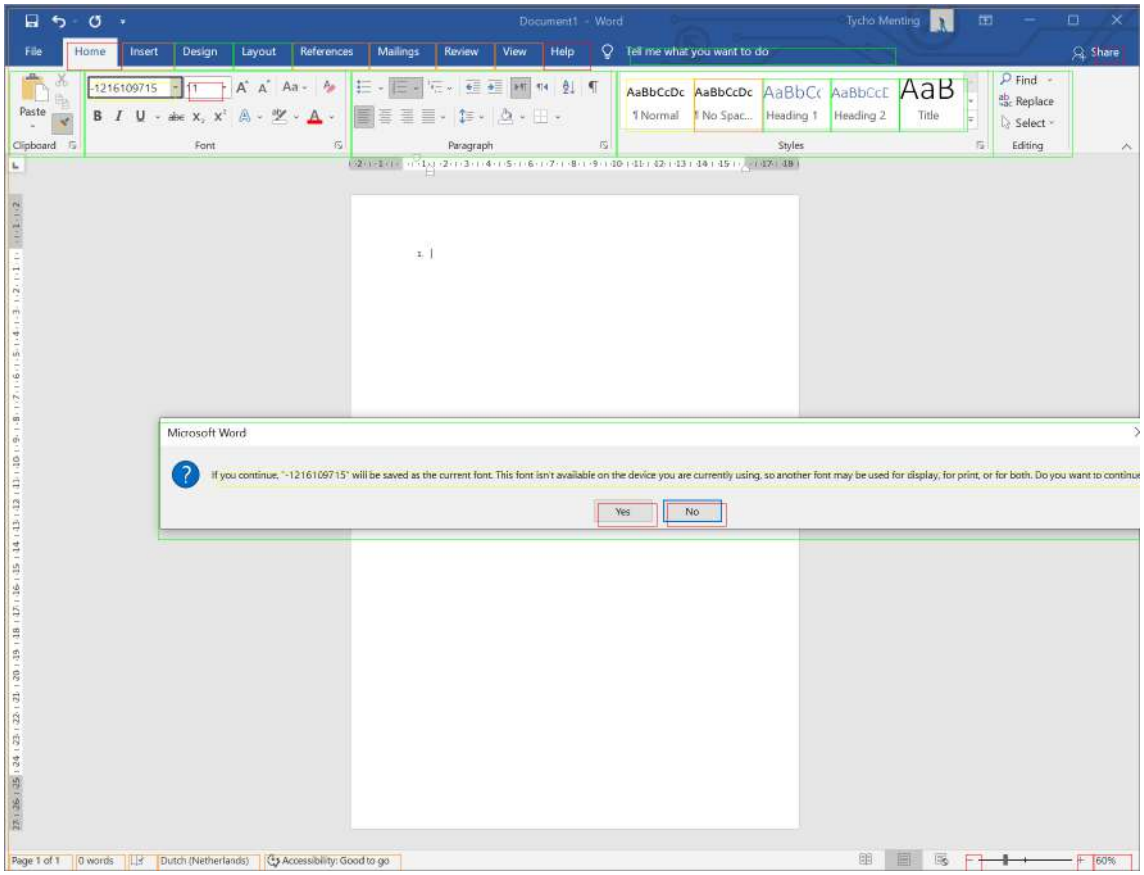


Figure 6.11: A cutoff dialog screenshot.

- We noticed that we have configured the applied filter in such a way that it leads to excluded text elements for state seven, while they should have been processed by our validation mechanism. As we can see in the screenshot 6.12, the text elements starting with `Bottom Border` until `Diagonal Up Border` have been excluded while they do contain visible text. During the test run, we have added the color picker pop-up to the ignore filter because the widgets only consisted out of a coloured rectangle. Unfortunately the text widgets for configuring the border match with this filter. Because of our current implementation of the filter mechanism, it is not possible to resolve this side effect.

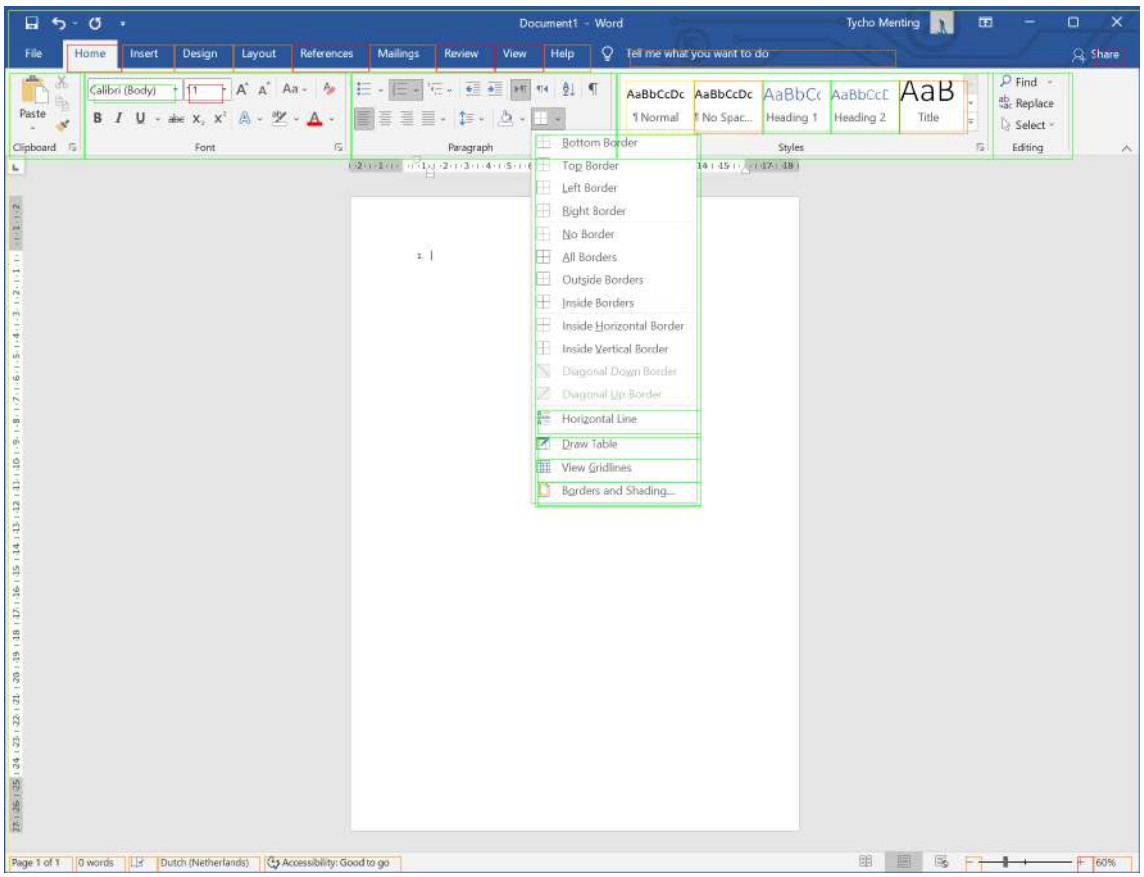


Figure 6.12: State seven contains unintended excluded text fields.

- Our implementation has highlighted the statusbar widgets in orange, as we can see, for example, in the bottom of screenshot 6.10. For example, the detected text `Page 1 of 1` needed to be matched with the expected text `Page Number Page 1 of 1`. Because this is applicable for all the widgets which are located on the statusbar it is most likely that the developer manipulates the presentation by either changing the size of the widget or filtering the text before showing the widget, or the expected text is not stored in the used "*Tag*". We have inspected the available "*Tags*" but could not find another one which stored the expected text correctly.

- While it is debatable whether statusbar widgets should be marked as a TPF, it is obvious that the following result is an actual TPF. Screenshot 6.13 shows the extended view of the `Styles` group. Within this view, our validation mechanism has detected the following seven TPFs:

  - Expected text `¶ No Spacin`, discovered text `No Spac..`
  - Expected text `¶ List Paragrap`, discovered text `List Para...`
  - Expected text `Intense Reference`, discovered text `Intense Re...`
  - Expected text `Subtle Reference`, discovered text `Subtle Ref...`
  - Expected text `Intense Quote`, discovered text `Intense Q...`
  - Expected text `Intense Emphasis`, discovered text `Intense E...`
  - Expected text `Subtle Emphasis`, discovered text `Subtle Em...`

  We assume that all seven widgets are configured to support ellipsis. When ellipsis is enabled, three dots will truncate the text when the text is too long for the available

space. Although our implementation can not detect whether such a feature is being used by the SUT, this a valid outcome. The expected text is not entirely visible for the user and therefore the result is valid.

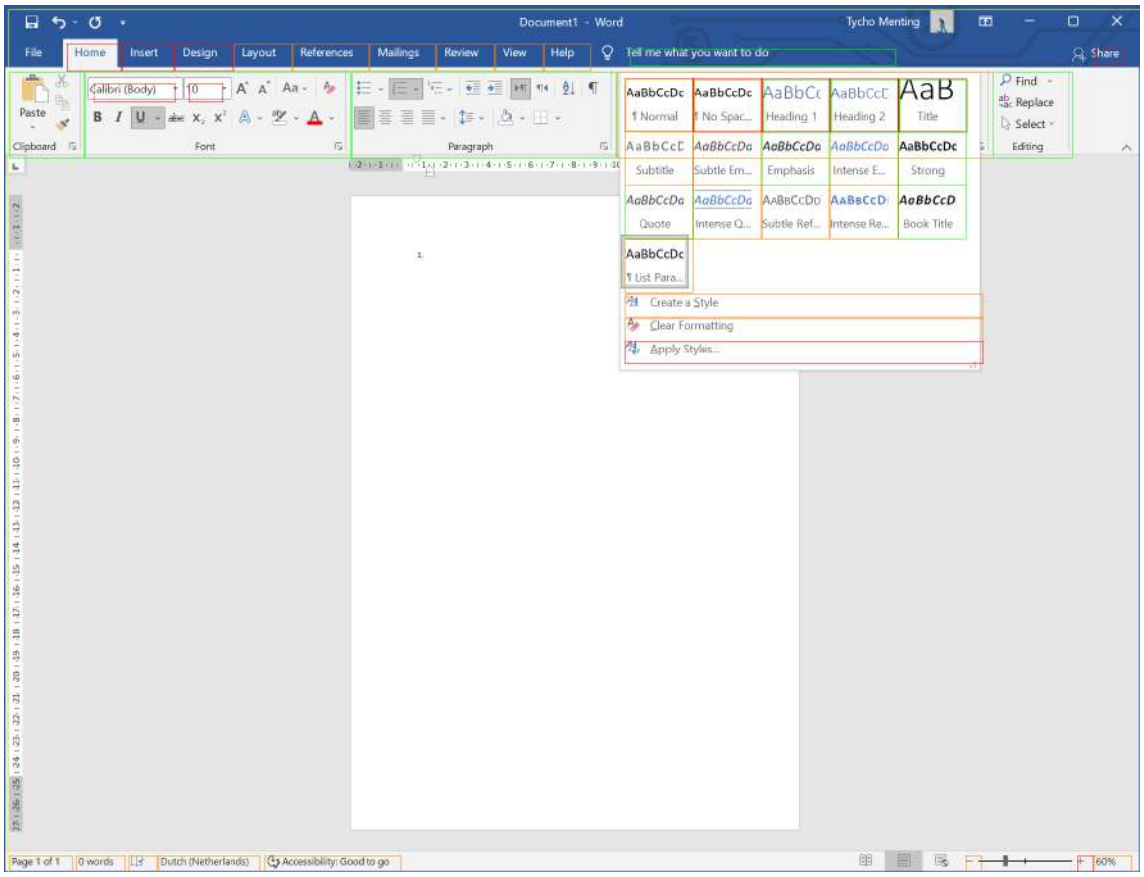


Figure 6.13: The last state containing many TPFs.

Screenshot No	Expected elements	Expected elements filtered (actual)	Excluded despite visible text	Covered by action	Not detected by OCR	TPF	Entire matches	Partially matched	
								OCR	Algorithm
1	166	28(26)	6	0	7	1[b]	9	7[c]	2
2	169	31(29)	6	0	4	5[bc]	11	8	1
3	1	0(0)	0	0	0	0	0	0	0
4	170	32(29)	7	0	4	5[bc]	15	4	1
5	1	0(0)	0	0	0	0	0	0	0
6	166	28(26)	7	0	6	3[bc]	9	6	2
7	1	1(1)	0	0	0	0	1	0	0
8	166	28(26)	7	0	6	2[bc]	8	7	3[c]
9	1	0(0)	0	0	0	0	0	0	0
10	177	36(33)	7	0	7	5[bc]	12	8[d]	1
12	1	1(1)	0	0	1	0	0	0	0
13	170	32(29)	7	0	4	5[bc]	15	4	1
14	1	(0)	0	0	0	0	0	0	0
15	179	39(34)	7	0	4	5[bc]	17	7	1
16	1	0(0)	0	0	0	0	0	0	0
17	192	37(33)	19	0	6	6[bc]	12	7	2
18	1	0(0)	0	0	0	0	0	0	0
19	170	32(29)	7	0	4	6[bc]	9	8	2
20	1	0(0)	0	0	0	0	0	0	0
21	170	32(29)	7	0	4	5[bc]	15	4	1
22	1	0(0)	0	0	0	0	0	0	0
23	196	51(42)	0	1	5	12[abc]	12	8	4

Table 6.3: Measurements Microsoft Word.

[a]

- Expected text ¶ `No Spacin`, discovered text `No Spac..`
- Expected text ¶ `List Paragrap`, discovered text `List Para...`
- Expected text `Intense Reference`, discovered text `Intense Re...`
- Expected text `Subtle Reference`, discovered text `Subtle Ref...`
- Expected text `Intense Quote`, discovered text `Intense Q...`
- Expected text `Intense Emphasis`, discovered text `Intense E...`
- Expected text `Subtle Emphasis`, discovered text `Subtle Em...`

[b] The paragraph character is missing and the expected text is `No Spacin` while `No Space..` is visible. [c] Does contain one or more "*UIAText*" located on the "*UIAStatusBar*", this widget

- Expected text `Word Count 0 words`, discovered text `0 words`
- Expected text `Page Number Page 1 of 1`, discovered text `Page 1 of 1`
- Expected text `Language Dutch (Netherlands)`, discovered text `Dutch (Netherlands)`
- Expected text `Accessibility Checker Accessibility: Good to go`, discovered text `Accessibility: Good to go`
- Expected text `Status Message Select content to apply the copied formatting, or press Esc to cancel`, discovered text `Select content to apply the copied formatting, or press Esc to cancel`
- Expected text `Zoom 60%`, discovered text `60%`

[d] Expect text `If you continue, "-1216109715" will be saved as the current font. This font is not available on the device you are currently using, so another font may be used for display, for print, or for both. Do you want to continue?` because of the screenshot dimensions, the dialog was not captured entirely.

Table 6.3 shows the measurements performed on the outcome of the run with Microsoft Word as SUT. Our implementation processed 2100 widgets in total during the run. After applying our filtering mechanism, we have validated 408 expected text elements. Inspecting the screenshots manually resulted that this should have been 365 text elements instead. This means that 11% of the expected text elements were processed by our implementation, while they should not have been. These text elements will end up as false positives in the report. On the other side, our filtering mechanism excluded 87 visible text elements. These visible text elements should have been analyzed by our implementation. This means that 19% of the text elements on the screenshot have not been analyzed. From the 365 actual expected text elements, 62 were not detected at all by the OCR engine. This is 17% of the widgets which needed to be analyzed. Also, another 21% has been detected partially by the OCR engine. While 5% could not be entirely matched, because of our matching algorithm. 16% of the expected text elements are correctly marked as TPF and 40% of the expected text has entirely been matched by our implementation.

6.4 Windows Media player

6.4.1 Composing the configuration.

While investigating which widgets needed to be excluded by our filtering mechanism, we encountered two cases which we want to elaborate.

1. As we can see in figure 6.14, most of the "*UIAButton*" widgets do not contain visible text. Therefore, we have decided to ignore them and filter them out. This would imply that we will ignore the following buttons despite they contain visible text.
   1.1 `Play`
   1.2 `Burn`
   1.3 `Sync`
   1.4 `Save list`

However, after investigating the remaining expected text elements, we noticed that first three buttons were still expected. After investigating this, we discovered that for each of these buttons, there was also an "*UIAEdit*" widget positioned at the same location. These widgets are well integrated into the button and therefore not noticeable. Because of this duplication, we can still analyze these expected text elements, because "*UIAEdit*" widgets are analyzed by their "*UIAValueValue*" tag. Only the `Save list` will be ignored by our implementation.

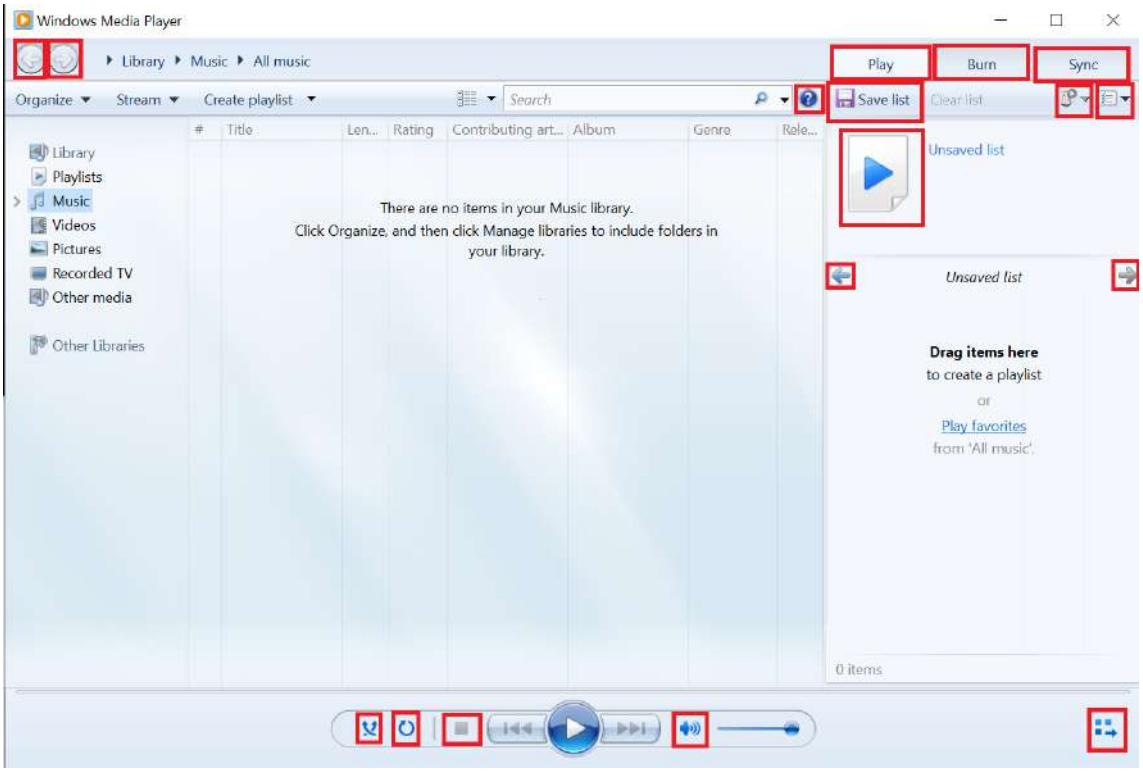


Figure 6.14: Windows Media player `UIAButtons`.

2. The main screen of Windows Media Player contains two widgets of the type "*UIASplitButton*". The first one has the visible text `Create playlist` and the second one contains no visible text and was called `View options`. This "*UIASplitButton*" without visible text is located left of the search box. Both buttons have the same ancestor path
   `::UIAToolBar::UIAPane::UIAPane::UIAPane::UIAWindow::Process`. Therefore it is not possible with the current filtering mechanism to exclude one of them. We have decided to not exclude the "*UIASplitButton*" widgets and only ignore the `View options` from the results.
3. For the widget type "*UIAMenuItem*" we have the same scenario as we have for "*UIASplitButton*". The widgets with text; `Organize` and `Stream` contain visible text while the element right of search box contains the hidden text `Search options`. Here we have also chosen not to exclude this widget and to ignore the invisible text while analyzing the results.

Because Windows Media Player contains many widgets which do not lead to a different state, we have extended the `ClickFilter` setting. This is a list of regular expressions. If the text from a widget matches with a regular expression, then the widget is not available as an action. We have used the following regular expressions:

- `*[sS]ystem.*`, prevents that we perform an unnoticeable action on the SUT.
- `.*[eE]xit.*`, prevents that we close the SUT.
- `.*[mM]inimi[zs]e.*`, prevents that the SUT is minimized.
- `Close`, prevents that we close the SUT.
- `Maximize`, prevents that we maximize the SUT.

The following regular expressions prevent that we perform an action which does not introduce new text elements.

- `Turn.*`
- `.*Glyph.*`
- `Mute`
- `.*Seek`
- `Volume`
- `.*Icon`
- `Switch.*`
- `csPlaystate`
- `.*Search.*`
- `Play`
- `.*Back.*`
- `.*Details Pane.*`
- `.*Library Navigation Pane.*`
- `.*Burn Drive.*`

6.4.2 Inspecting the results.

We have chosen to configure ten actions and a single sequence for testing Windows Media Player as SUT. By executing ten actions on the SUT, we see enough visual changes to perform our measurements and collect the results for our research. Looking at the results, we see in general that most of the expected text elements are correctly identified by the OCR engine and that our matching algorithm successfully matches the entire expected text. Screenshot 6.15 shows the initial state of the Windows Media Player as SUT. We can see that from all the expected text elements, the majority is successfully detected by the OCR engine and all the characters have been completely matched. We will now elaborate on the ones that are not green or do not have an outline at all.

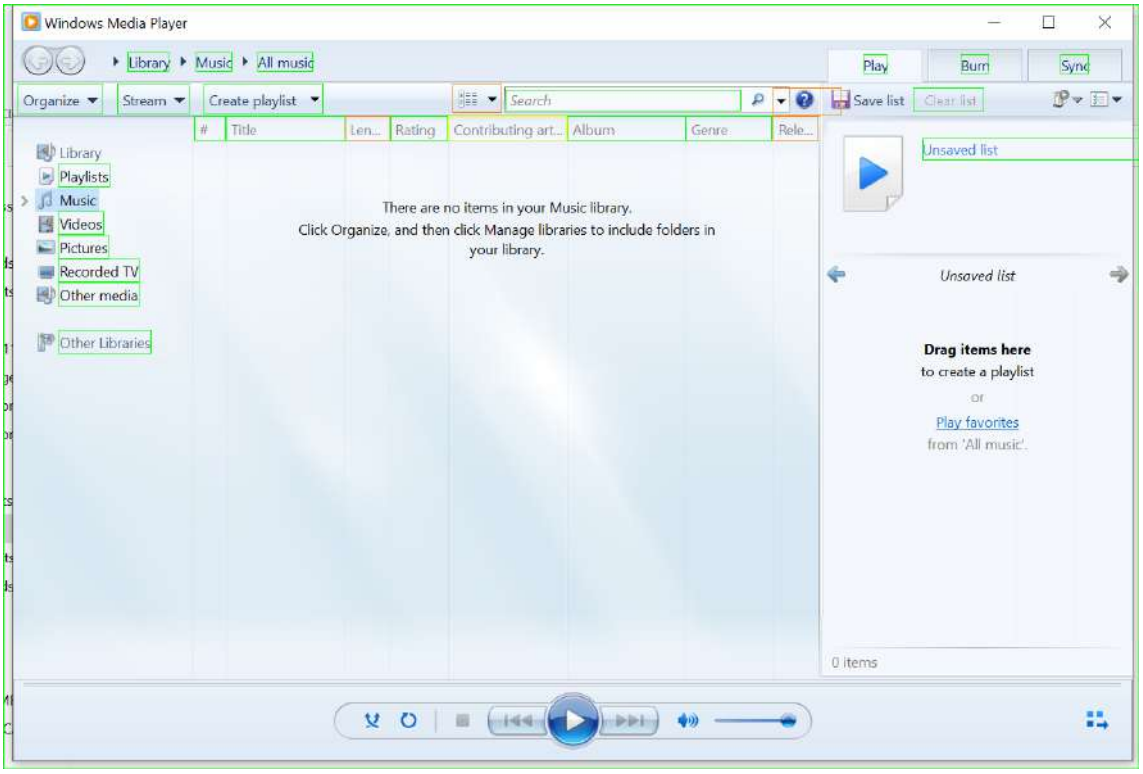


Figure 6.15: Windows Media player initial state.

- Screenshot 6.16 shows the "*UIASplitbutton*" and "*UIAMenuItem*" case, as discussed in the previous section. On the left side of the `Search` box, we see the "*UIASplitButton*" with the expected text `View options`. On the right side, we see the "*UIAMenuItem*" with the expected text `Search options`. We have excluded both widgets while performing the measurements.

Figure 6.16: The included UIASplitButtons in Windows Media player.

- Screenshot 6.17 contains a lot of text elements without an outline. The widget with the text `Unsaved list` was from an "*UIAPane*" type. This type has been ignored because the majority of them did not contain visible text. The other visible text elements in the screenshot are "*UIAList*" widgets. The "*Title*" tag did not contain visible text and after manually inspecting the remaining tags we could not find a match with the visible text. Therefore, we have decided to ignore the "*UIAList*" widgets as well.

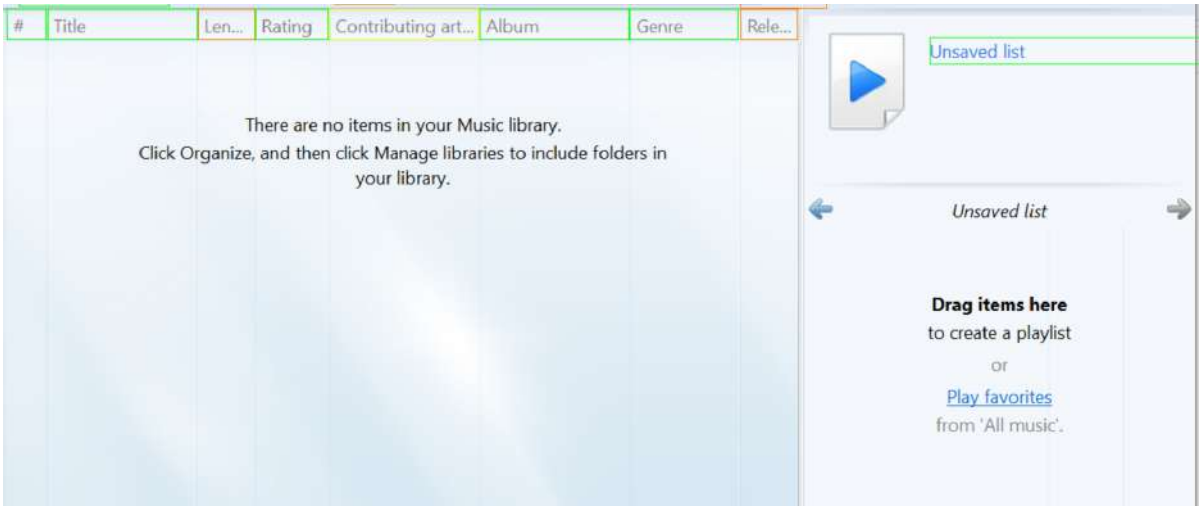


Figure 6.17: Excluded text in Windows Media player.

- Screenshot 6.18 contains a bug in our implementation. For unknown reasons, the outline is missing in the screenshot. We have inspected the log files and noticed that both `Library` and `Music` have been matched 100%. They should have been outlined in green, similar to the other text elements in the screenshot.

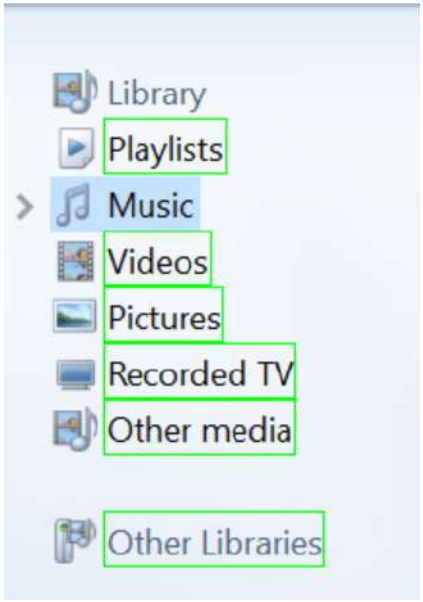


Figure 6.18: The missing outline for visible text in Windows Media player.

- Screenshot 6.19 shows three correctly detected TPFs. For the visible text element `Len...`, the expected text is `Length`. For the visible text element `Contributing art...`, the expected text is `Contributing artist`. And for the last visible text element *Rele...*, the expected text is `Release Year`. For all three text elements, the text has been truncated and three dots have been added.

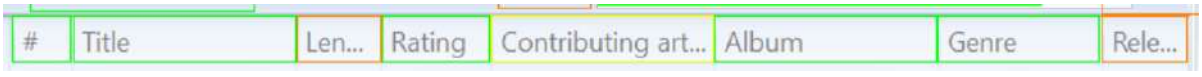


Figure 6.19: Valid TPFs detected in Windows Media player.

- Screenshot 6.20 shows a pop-up menu which covers the expected text element `Create playlist`. The executed action has triggered this pop-up menu. Because the expected text is covered, the OCR engine can not detect the expected text. Also, some of the expected text elements from the pop-up menu are linked to the covered element because the location of the detected text intersects with the expected text.

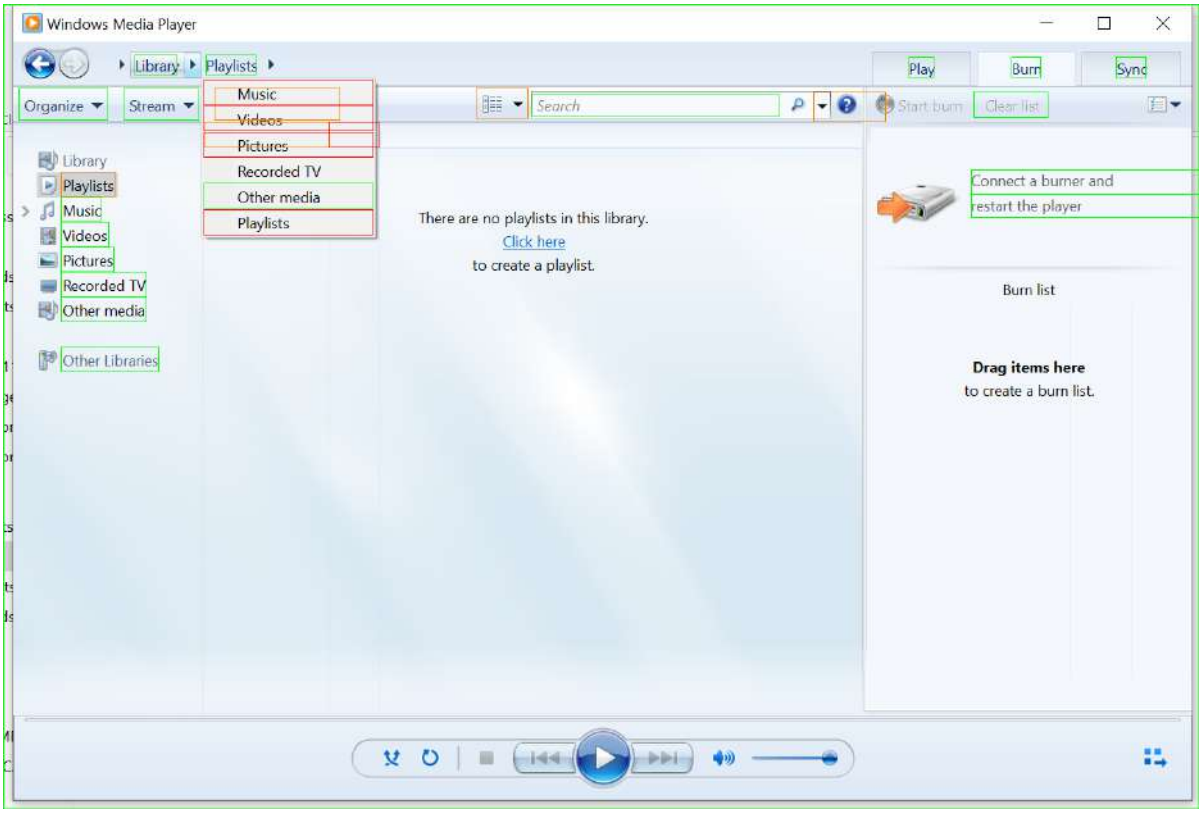


Figure 6.20: Expected text covered by a pop-up in Windows Media player.

- In screenshot 6.21, we see that the `Search` widget has been outlined as if it was entirely matched with the OCR results. However, after manually inspecting the output, we have discovered that the OCR engine did not detect any text for this expected text element. Therefore, the given outline color is incorrect and should have been red instead of green.

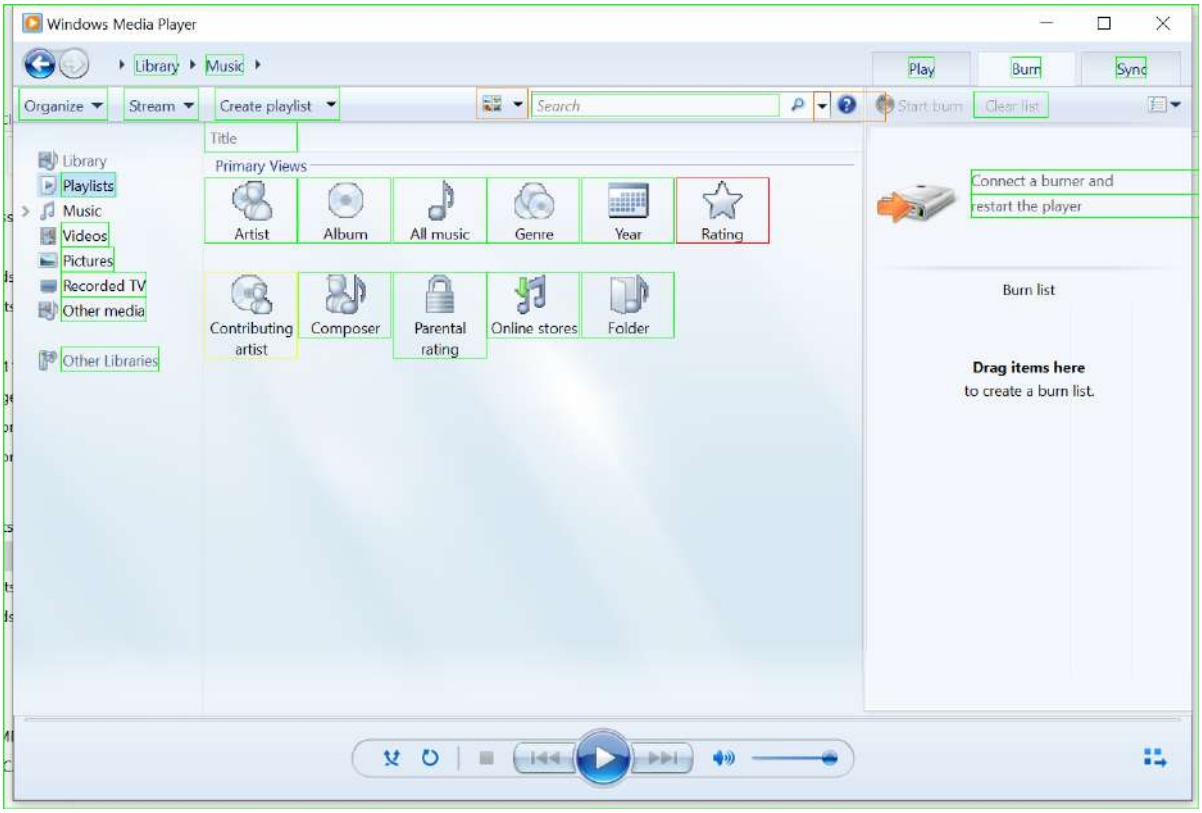


Figure 6.21: `Search` has incorrectly been marked as entire match.

Screenshot No	Expected elements	Expected elements filtered (actual)	Excluded despite visible text	Covered by action	Not detected by OCR	TPF	Entire matches	Partially matched	
								OCR	Algorithm
1	152	$31^a(29^b)$	4	0	0	3	26	0	0
2	152	$31^a(29^b)$	4	0	0	3	26	0	0
3	1	0(0)	1	0	0	0	0	0	0
4	154	$34^a(32^b)$	4	0	0	0	28	4	0
5	1	$0^a(0)$	1	0	0	0	0	0	0
6	149	$35^a(33^b)$	4	0	0	0	29	4	0
7	1	$0^a(0)$	0	0	0	0	0	0	0
8	149	$35^a(33^b)$	4	0	1	0	31	1	0
9	1	$1^a(0)$	0	0	0	0	0	0	0
10	149	$35^a(33^b)$	4	0	2^c	0	30	1	0
11	1	$1^a(1)$	0	0	0	0	1	0	0
12	149	$35^a(33^b)$	4	0	0	0	32	1	0
13	1	$1^a(1)$	0	0	0	0	1	0	0
14	143	$28^a(26^b)$	5	0	1	0	23	2	0
15	1	$1^a(1)$	0	0	0	0	1	0	0
16	130	$25^a(23)$	0	0	1	0	22	0	0
17	1	$1^a(1)$	0	0	0	0	1	0	0
18	140	$27^a(24)$	4	0	1	0	21	2	0
19	1	$0^a(0)$	0	0	0	0	0	0	0
20	145	$31^a(26)$	4	3	1	0	21^b	1	3
21	1	$1^a(1)$	0	0	0	0	1	0	0
22	149	$35^a(33^b)$	0	0	1	0	29	3	0

Table 6.4: Measurements Windows Media Player.

[a] Includes a widget "*UIASplitButton*" with text `View options` and a widget "*UIAMenuItem*" with the text `Search options`
[b] Contains one or more of the following text elements; `Recorded TV`, `Other media`, `Library` or `Music` which are detected and 100% matched correctly but not annotated with a green border.
[c] `Search` has been marked as successful while the OCR could not find the text.

Table 6.4 contains the measurements extracted from the report. We excluded the first screenshot from the overall calculation. This is because the state was a duplication of the second screenshot. If we would include the first state, then the absolute numbers for the different categories would not represent the actual state. With Windows Media Player as SUT, our implementation has received 1619 widgets. Out of this input, we have automatically filtered 356 widgets, which we wanted to analyze. After a manual verification, we concluded that 29 of the expected widgets did not contain readable text. This means that there are 327 text elements which actually contain visible text. This comes down to 8% of the filtered widgets should not been analyzed by our implementation. 39 visible text elements were not included. This is 11% of all the text elements that were visible during the run. The OCR engine did not detect 8 visible text elements. This is 2.4% of the actual expected text elements. 3 expected text elements could not be detected because another widget covered them. 3 out of the 327 analyzed text elements were correctly detected as TPFs. 91% of the actual expected text elements are entirely matched with the OCR results. We could not entirely match 3 expected text elements because of the current implemented matching algorithm. While, another 19 expected text elements were only

partially detected by the OCR engine.

6.5 www.ou.nl

6.5.1 Composing the configuration.

We noticed during the exploration run that TESTAR could not execute the selected action correctly. We have encountered this problem during the proof of concept 3.1. This problem is computer dependent and therefore it is not always required to change this value. We have solved the problem for our setup by setting the display scale value to `1.0`. Except for the display scale, there was no need to modify other default settings. Also, expanding the configuration for ignoring certain widget types was unnecessary.

6.5.2 Inspecting the results.

Navigating through the SUT showed enough diversity in the visible text elements. Therefore, we have chosen to run a single sequence consisting out of ten actions. In screenshot 6.22, we see the initial state of the SUT. The OCR engine has detected the text within the logo and the search icon while they were not expected. The OCR engine has identified them as `Open Universiteit` and the character `Q`. The expected text elements in the upper right corner of the screenshot were not detected at all by the OCR engine. The same applies to the text `Groei. Durf. Studeer. #DenkOpen`. The OCR has detected one character incorrectly for the expected text `Nu is jouw moment`. The identified text was `Nuts` instead of the expected text `Nu is`. While the OCR engine has identified the entire text of the cookie banner at the bottom of the screenshot correctly, our matching algorithm was not capable of matching all the characters correctly. Our implementation has identified and matched the remaining eleven expected text elements correctly. While inspecting the remainder of the report, we noticed the following items:

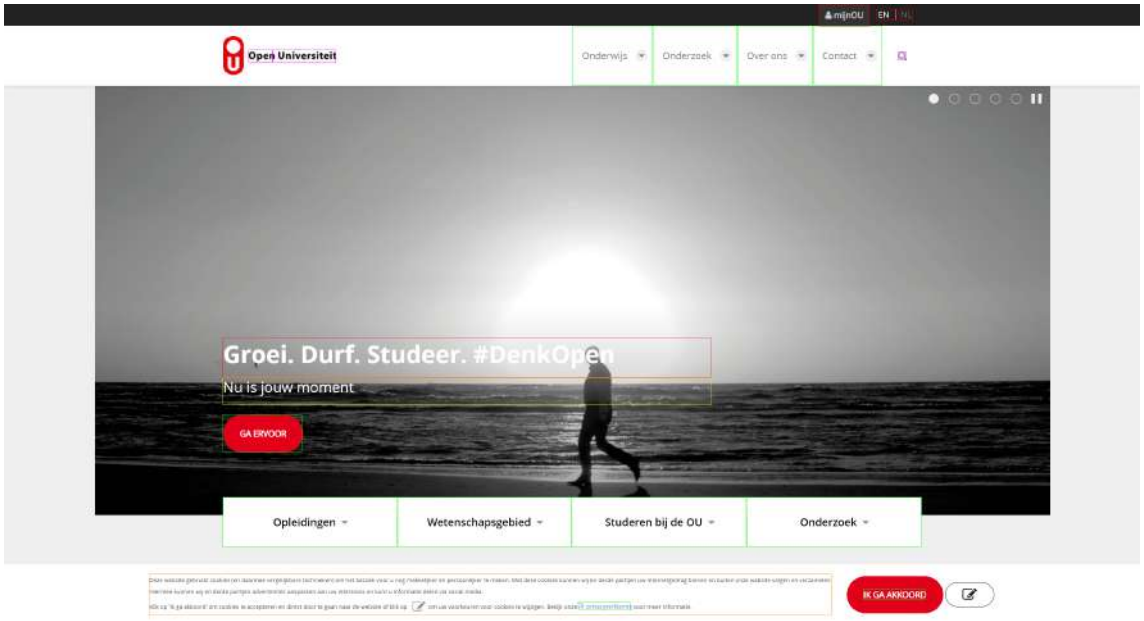


Figure 6.22: The initial state of the OU website.

- Screenshot 6.23 contains an image at the bottom of the screenshot. This image contains text which is detected correctly by the OCR engine. However, all the text incorporated into images can not be extracted by the accessibility API and are therefore not included in the expected text list.

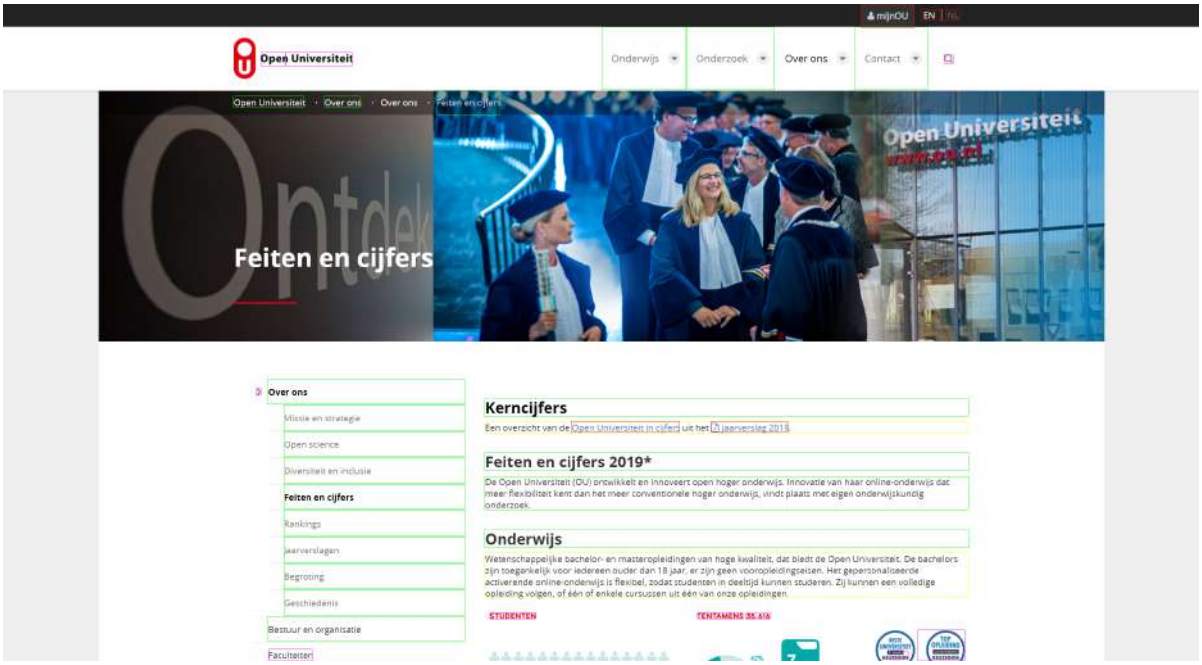


Figure 6.23: Images on the OU website which contain text.

- Screenshot 6.24 shows a red line below the expected text `Contact en adressen`. This red line is not an annotation by our implementation. Instead, it is a floating element on top of the background image, which is part of the website design.

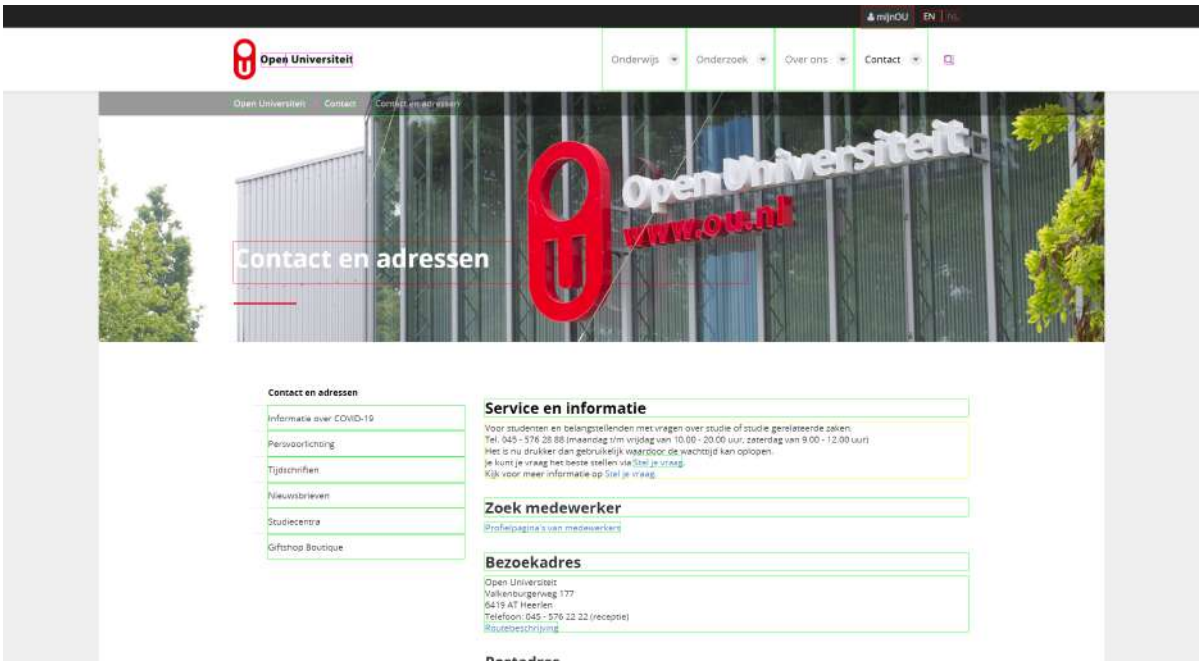


Figure 6.24: A screenshot of the OU website containing an unexpected red line.

- Screenshot 6.25 shows a part of the website which contains an embedded video. The embedded video contains visible text, which is detected by the OCR engine. The text elements where not included in the expected text list. This is because the video is included as an `Iframe`. The current implementation can not query content within the `Iframe`.

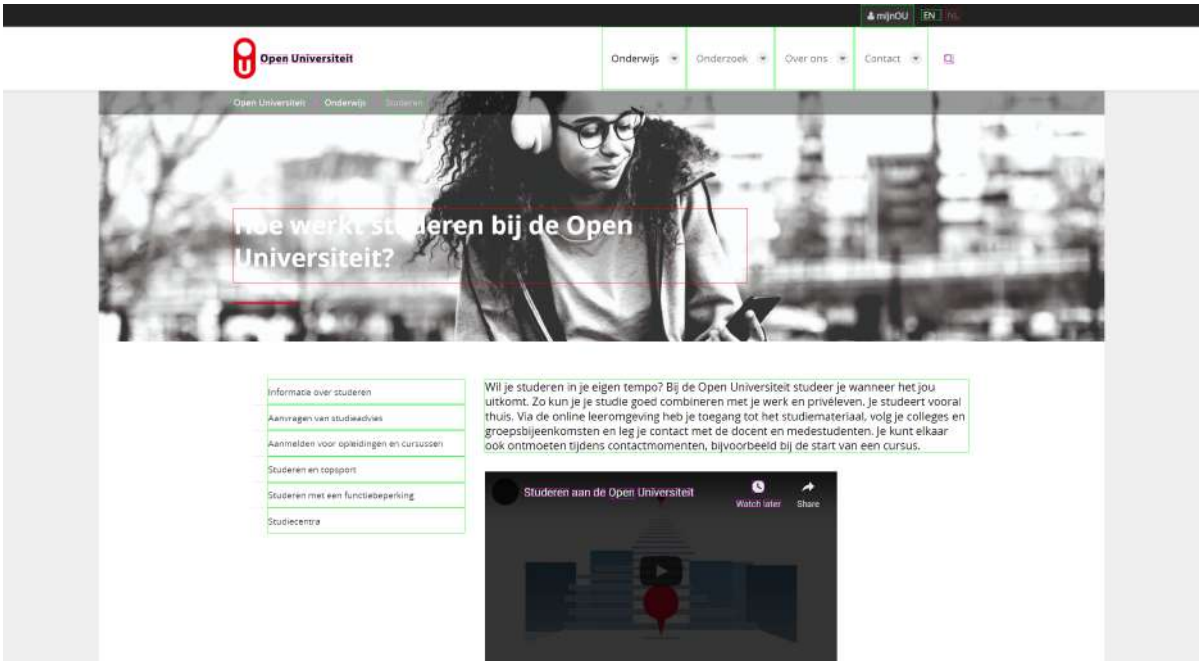


Figure 6.25: An `Iframe` on the OU website containing an embedded video.

- We see at the bottom of screenshot 6.26 a single sentence which comprises two outline colors. The first outline is a purple and located on the left and right edge. The second outline is a green and located in the center of the sentence. This sentence is built out of two parts, a hyperlink in the middle with the expected text `acht wetenschapsgebieden` and the remainder of the sentence. The hyperlink has successfully been detected by the OCR engine and correctly matched. As for the other part, the OCR engine did correctly detect the individual words. However, the original text is longer than we can see on the screenshot. When we extract the expected text, we have included an additional check for the "*webdriver protocol*". Websites often contain more context than the web browser can present on the screen. Therefore, users need to scroll through the page to see the remainder of the website. When we extract the expected text, we only include expected text items which are visible by checking if the web tag "*WebIsFullOnScreen*" is set to true. In this example, it is not the case.

Figure 6.26: Partially visible text at the bottom of the screenshot.

- Screenshot 6.27 contains a chat window on the right side of the screenshot. The text elements within this window are all annotated with a purple outline. Similar as in the previous case 6.5.2, the chat window is placed within an `Iframe`. This means that we can not extract the expected text elements with our current implementation.

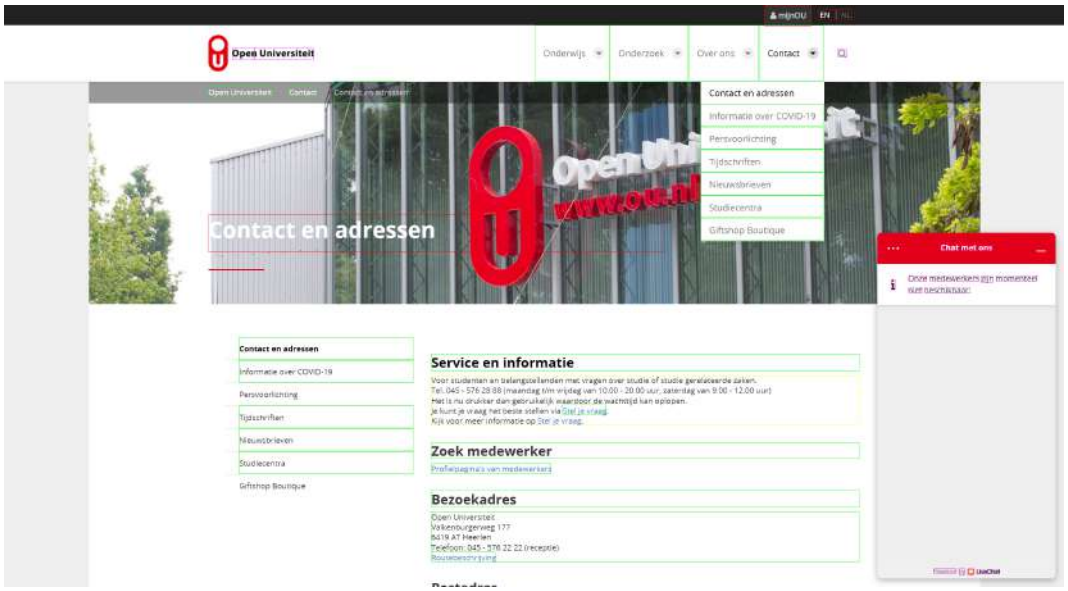


Figure 6.27: A chat window embedded as `Iframe`.

Screenshot No	Expected elements	Expected elements filtered (actual)	Excluded despite visible text	Covered by action	Not detected by OCR	TPF	Entire matches	Partially matched	
								OCR	Algorithm
1	559	17(17)	0	0	4	0	11	1	1
2	558	17(17)	0	0	4	0	10[a]	2	1
3	1	1(1)	0	0	0	0	1	0	0
4	545	14(14)	0	0	4	0	8[a]	2	0
5	1	1(1)	0	0	0	0	1	0	0
6	403	12(12)	0	0	1	0	9	2	0
7	1	1(1)	0	0	0	0	1	0	0
8	255	27(27)	0	0	2	0	23[a]	1	1
9	1	1(1)	0	0	0	0	1	0	0
10	271	30(30)	0	0	5	0	23[a]	2	0
11	1	0(0)	0	0	0	0	0	0	0
12	280	34(34)	0	0	5	0	27	2	0
13	1	1(1)	0	0	0	0	1	0	0
14	259	18(18)	0	0	2	0	16	0	0
15	1	0(0)	0	0	0	0	0	0	0
16	544	14(14)	0	0	4	0	8[a]	2	0
17	1	0(0)	0	0	0	0	0	0	0
18	544	14(14)	0	0	5	0	8[a]	1	0
19	1	1(1)	0	0	0	0	1	0	0
20	292	27(27)	0	0	4	0	22	0	1
21	1	0(0)	0	0	0	0	0	0	0
22	309	34(34)	0	0	4	0	29[a]	0	1

Table 6.5: Measurements for www.ou.nl

[a] Missing green outline for: `Onderzoek`, `Over ons`, `Feiten en cijfers`, `Onderzoek`, `Contact en adressen`, `Stel je vraag`, `Persvoorlichting` and `Giftshop Boutique`.

Table 6.5 shows the measurements for www.ou.nl as SUT. Our implementation has processed 5389 elements in total. After feeding the widgets to our filtering mechanism, we needed to validate 264 expected text elements. After manually inspecting the screenshots, we conclude that all text elements which needed to be analyzed are included in the outcome of the filtering mechanism. And that the filter elements are all valid and do not contain any unexpected text elements. Compared to the desktop applications, we see that the filtering mechanism needed to filter significantly more elements. Also, the entire filtered output consists out of true positives. This is something we have not seen before, not even in the reference application. None of the expected text elements were blocked because of an executed action. So all the expected text elements are visible for the OCR engine. The OCR engine could not detect 44 text elements from the total of 264 expected text elements. This is 16% of the text elements which needed to be validated by our implementation. There are no TPFs identified while testing the website. Our implementation has successfully identified and matched 200 expected text elements. This is 76% of total expected text elements. 15 text elements could only partially be matched because of incorrect detection by the OCR engine. This is 6% of the total expected text elements. 5 other text elements could only partially be matched because of our implemented matching algorithm.

6.6 www.nytimes.com

6.6.1 Composing the configuration.

Similar as for the www.ou.nl website and during the proof of concept 3.1 we changed the display scale to 1.0 for testing www.nytimes.com. The www.nytimes.com website presented a cookie banner at the bottom of the website. Screenshot 6.28 shows this cookie banner. Initially, we tried to enforce that the initial action would remove this banner. This has been done similarly for the www.ou.nl website, where the protocol has specified a policy to close the banner. We tried to implement a policy based on the class. However, while testing this policy, we noticed that the action remained available even when the banner was gone. Therefore, all the execute actions led to clicking on a none visible button. This behavior did not lead to new states of the SUT. We decided to bypass the cookie banner. Because of legislation, the cookie banner is only shown when we requested the website from within Europe. In order to bypass this, we needed to virtually move to outside Europe. Therefore, we used a VPN connection with the USA. After opening the website with the VPN enabled, we noticed that the cookie banner was not shown anymore. The default configuration did not need to be modified, therefore we have used it as is while using www.nytimes.com as SUT. While composing the configuration, we have noticed that the website offers many different text elements when executing a single sequence with ten actions. Therefore, we have decided to use this setup for the actual run, which will be used to measure the performance of our implementation.

Figure 6.28: The cookie banner shown on The New York Times website.

6.6.2 Inspecting the results.

Screenshot 6.29 shows the initial state of the website. As we can see in table 6.6, the majority of the expected has been matched either entirely or almost entirely correct with the detected text. Right above the text `Thursday, February 3, 2022`, we see a red outline. We did not take this into account as expected text element. The website contains two `anchor`s at this position with the expected text `Skip to site index` and `Skip to content`. There was no possibility of filtering this out without ignoring many visible text elements. The expected text `Opinion` located at the bottom left of the logo does not contain an outline. After inspecting the log files, we noticed that the expected text element was correctly identified and match and that only the outline was not drawn correctly. After inspecting the report, we want to highlight the following items:

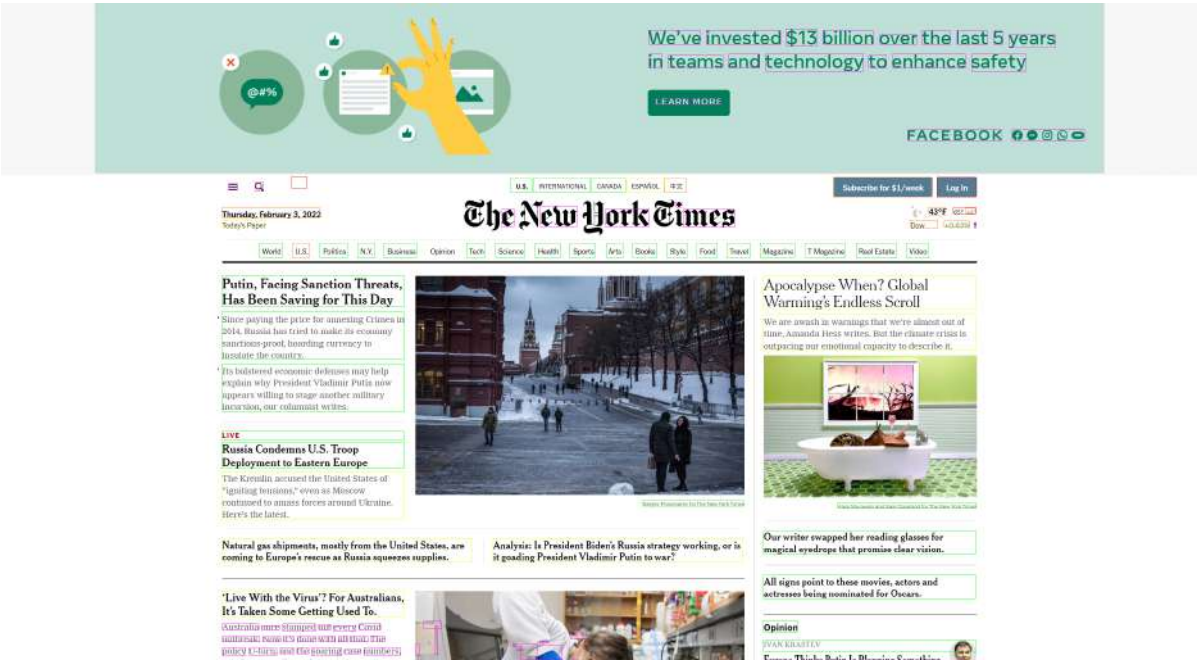


Figure 6.29: The initial state of The New York Times website.

- Screenshot 6.30 contains an advertisement shown on top of the website. These advertisements are placed within an `Iframe`. It is not possible with the current implementation to extract the expected text for content shown within this `Iframe`.

Figure 6.30: Text in advertisements can not be extracted.

- Screenshot 6.31 shows stock information below the weather information. The stock information is replaced with new stock information after a brief period of time. The screenshot capture and the extraction of the expected text do not take place at the exact same moment. Therefore, it is possible that the captured screenshot contains the previous stock information and that the expected text already contains the new stock information. Our implementation can not handle this type of animated content. This can lead to a mismatch between identified and expected text.

Figure 6.31: Animated stock information is not detected correctly.

- Screenshot 6.32 contains minuscule red dots positioned in the upper left corner of each image. These red dots are anchors with the expected text `Photo` which our implementation could not successfully match. We could not exclude these anchors without excluding

other visible text elements. Therefore, we have ignored them while performing the measurements listed in table 6.6.

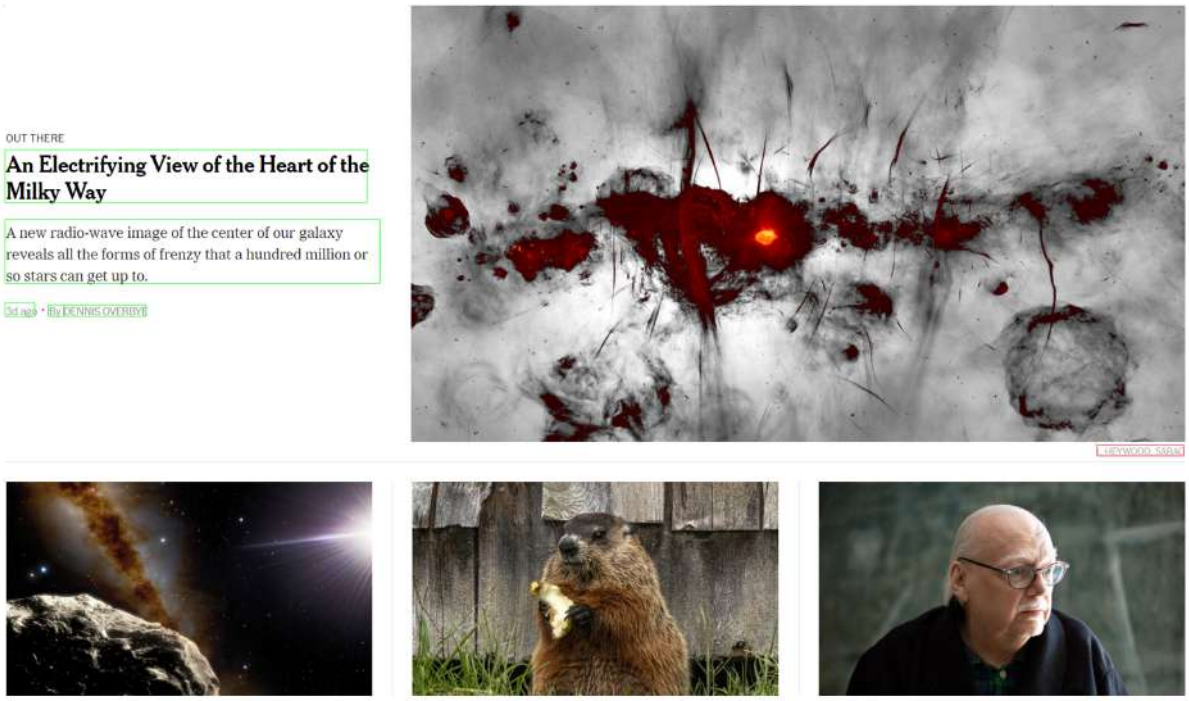


Figure 6.32: Image anchors which could not be excluded by the filter mechanism.

- Screenshot 6.33 shows many misaligned outlines. While the OCR engine could identify the visualized text based on the expected location they were not correctly matched. Manual investigation of this problem did not provide useful information to address this problem.

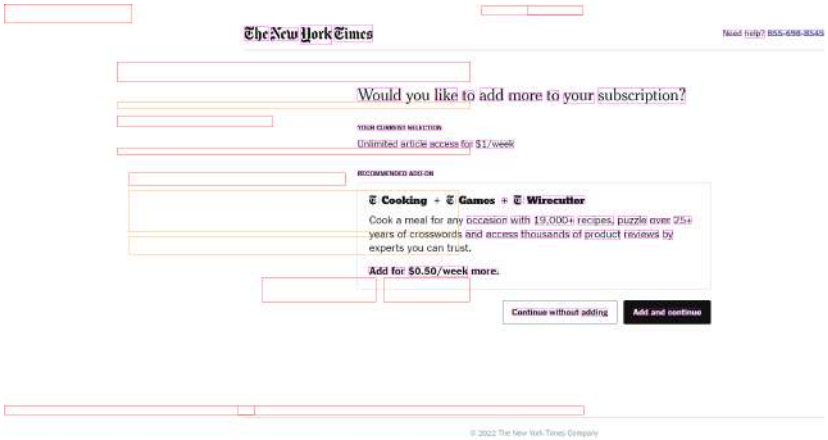


Figure 6.33: Expected text is misaligned with the OCR results.

- Screenshot 6.34 contains a pop-up requesting the user to log in or to create an account to be able to continue reading articles on the website. This pop-up is also placed within an `Iframe`, similar as for the advertisements 6.6.2 shown on the website. Because of the

`Iframe` we can not extract the expected text, despite that the content within the `Iframe` belongs to the website itself. Where the advertisement is outsourced.

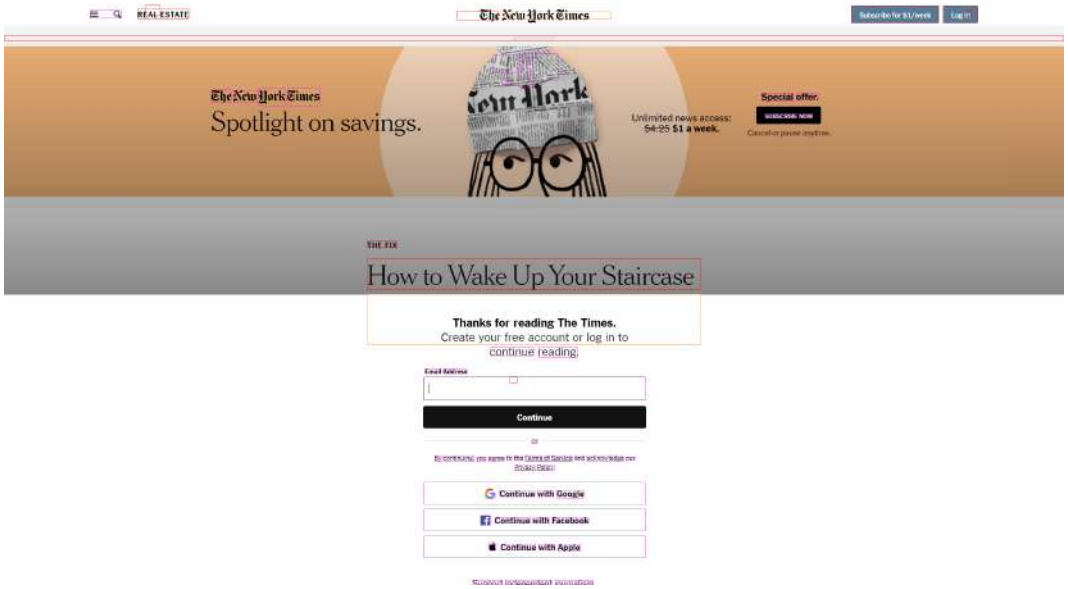


Figure 6.34: Text in popup is not expected.

- Screenshot 6.35 shows a yellow and orange outlined border. The OCR engine could not identify the expected Chinese characters for the orange outline. For the yellow outline, the OCR engine identified the text `ESPANOL` instead of the expected text `ESPAÑOL`. The tilde above the N was not detected. We noticed that the OCR engine has problems with identifying special characters like these. Also, for the initial state, as shown in screenshot 6.29, the remaining six yellow outlines are because the OCR engine did not detect characters like `"` and `’`.

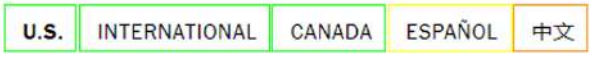


Figure 6.35: Special characters are not detected correctly by the OCR engine.

- Screenshot 6.36 contains a long text shown in state 12 of the SUT. Despite the good OCR results, our implementation could not entirely match this text correctly. The screenshot contains three expected text elements; the long text, `You can anytime.` and `cancel`. Our matching implementation should have assigned the word `introductory` to the long text. However, based on the location it was assigned to the expected text `You can anytime.` Because of this, our matching implementation tried to find the individual characters for the word `introductory` in the remainder of the long text. The correctly identified words were incorrectly assigned and could therefore not be assigned to their expected counter part.

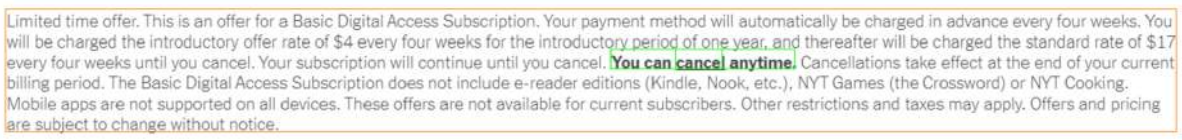


Figure 6.36: Incorrect location match for long text.

Screenshot No	Expected elements	Expected elements filtered (actual)	Excluded despite visible text	Covered by action	Not detected by OCR	TPF	Entire matches	Partially matched	
								OCR	Algorithm
1	1442	54[a](50)	0	0	2	0	32[b]	16	0
2	1442	54[a](50)	0	0	2	0	32[b]	16	0
3	1	1(1)	0	0	0	0	1	0	0
4	648	36[a](27)	0	0	5	0	21[b]	1	0
5	1	0(0)	0	0	0	0	0	0	0
6	641	16[a](6)	0	1	5	0	0	1	0
7	1	1(0)	0	0	0	0	1	0	0
8	648	36[a](27)	0	0	12	0	12[b]	1	2
9	1	1(1)	0	0	0	0	1	0	0
10	394	34[a](25)	0	0	4	0	20[b]	0	1
11	1	1(1)	0	0	0	0	1	0	0
12	382	40[a](27)	0	0	4	0	15	5	3
13	1	1(1)	0	0	0	0	0	1	0
14	118	22(22)	0	0	1	0	15	5	1
15	1	0(0)	0	0	0	0	0	0	0
16	1442	54[a](50)	0	0	4	0	33[b]	13	0
17	1	1(1)	0	0	0	0	1	0	0
18	594	28[a](18)	0	0	4	0	13	0	1
19	1	1(1)	0	0	0	0	1	0	0
20	118	22(22)	0	0	1	0	15	5	1
21	1	0(0)	0	0	0	0	0	0	0
22	34	14(14)	0	0	13[c]	0	0	0	1

Table 6.6: Measurements for www.nytimes.com

[a] Contains one or more anchors without visible text.
[b] Green outline is missing for `Opinion`, `WHAT YOU GET`, or `By`.
[c] OCR identified most text elements, however the location could not be matched with the expected element.

Table 6.6 presents the performed measurements on the website `www.nytimes.com` as SUT. Our implementation has processed 7913 widgets in total. According to our filtering implementation, 417 of them contain visible text. After a manual inspection, we counted 344 visible text elements. This means that 18% of the expected text elements should not be validated and will negatively affect the overall outcome for this test. None of the visible text elements were ignored because of our used configuration. Only one text element was covered by executing an action and could therefore not be identified by the OCR engine. The OCR engine has not identified 57 expected text elements while they were visible. We have not encountered a valid TPF while testing the website. Our implementation came to the same conclusion. 214 expected text elements have successfully been identified by the OCR engine and are fully matched. This is 62% of the overall expected text elements. 10 expected text elements could only partially be matched because of our matching implementation. The OCR engine has identified 64 text elements not entirely correct. Therefore, 19% could only partially be matched with the expected text.

6.7 www.apple.com

6.7.1 Composing the configuration.

While composing the configuration by performing dry-runs with the www.apple.com, we noticed that the selection algorithm often selected a drag action. Most of these drag actions did not result in new text elements which we can use to validate our implementation. Therefore, we have decided to disable the sliding actions. We have accomplished this by modifying the `Protocol_webdriver_generic.java`. By removing the function call "*addSlidingAction*" in the function "*deriveActions*", drag actions are no longer available as actions which can be executed on the SUT. We also noticed that the website uses a lot of animations when navigating between pages. The screenshot was often taken while the animation was ongoing. This resulted in expected text which could not be found by the OCR engine. In order to solve this, we have increased the "*TimeToWaitAfterAction*" setting from 0.1 till 10.0 seconds.

6.7.2 Inspecting the results.

Screenshot 6.37 shows the initial state of www.apple.com. The majority of expected text elements have not been matched correctly because the OCR engine could not detect them properly. Screenshot 6.38 shows the header containing the product overview of the website. The OCR engine has identified four items correctly. However, only `Airpods` could be linked to the expected text. This is because the expected text is extracted from an `Anchor` element. These elements have a height and width of only one pixel. Because of the dimensions, the outline is a single pixel. The outline for the header elements is placed at the location of the `Anchor` element. This is horizontal centered at left side for each item. While our matching implementation was not able to link all the discovered text elements correctly to the expected ones, it is remarkable that only four of the eleven expected text elements have been identified by the OCR engine, while the visual appearance across the text elements is the same.

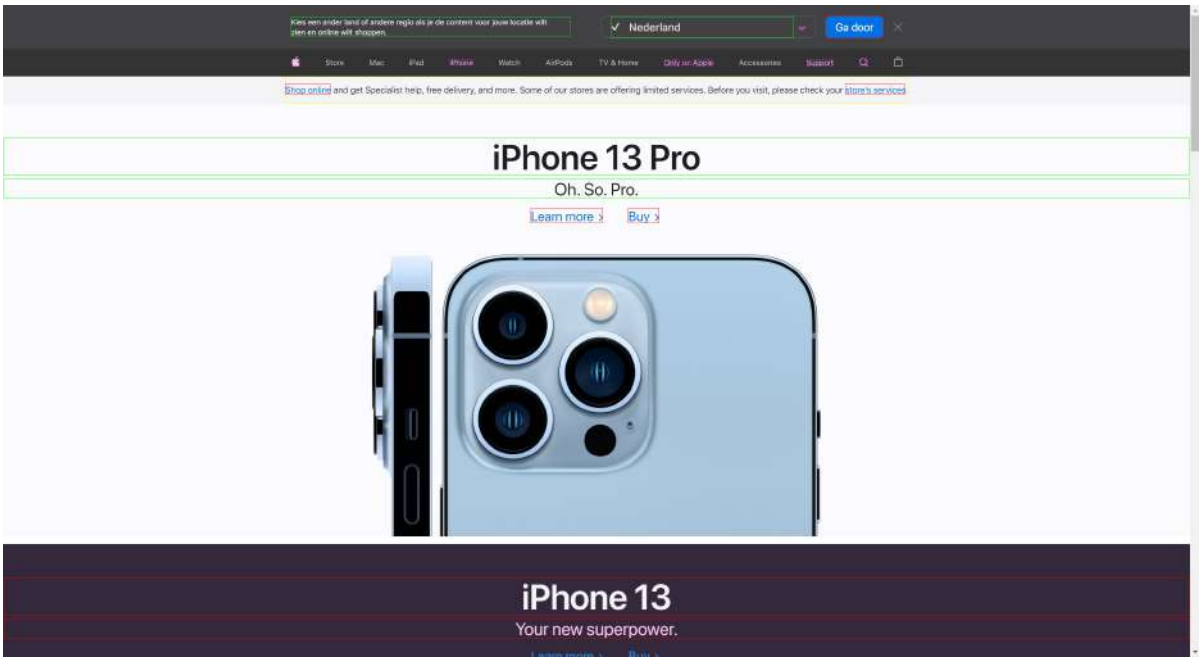


Figure 6.37: The initial state of the Apple website.

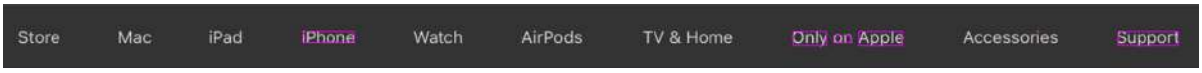


Figure 6.38: The product overview of the Apple website.

- Screenshot 6.39 shows a part of state 6. The expected text elements are `Inspired by Afrofuturism, the new Black Unity watch band and matching Unity Lights watch face symbolize the necessity for a more equitable future for the present generation of Black people` and `Black Unity Braided Solo Loop`. The OCR engine successfully detected these text elements entirely. However, our implementation did not mark this expected text element as entirely matched correctly. After inspecting the log files, we discovered that `Black Unity`, `Unity Lights` and `Black people` are separated by the `NO-BREAK SPACE` (U+00A0) character. Our matching algorithm ignores whitespaces, however, this character does not return true on the used check "*isWhitespace*". Screenshot 6.40 shows a part from state 14. This is a similar case where the `NO-BREAK SPACE` character is placed between `hits home`.

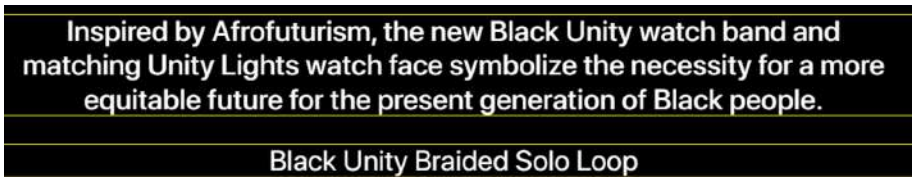


Figure 6.39: Text shown on the website of Apple contains invisible characters.

Figure 6.40: Text shown on the website of Apple contains invisible characters.

- Screenshot 6.41 contains a fragment from state 8. The expected text consist out of two elements: first `Get 6 months of Apple Music free with your AirPods.`. Second, `*`. The OCR engine has identified all the expected characters correctly. However, the text `Airpods.*` are all, based on their location, assigned to the `*`. Therefore, it was not possible to match the first section entirely and the text `Airpods.` remain unmatched.

Figure 6.41: Text was not matched entirely because of incorrect OCR assignment.

- State 10 is reached after the action to click on the search box was executed. Screenshot 6.42 shows a part of the state after the action was executed. The expanded search box covers the headers 6.38. Because of this overlap, the characters from the expected text `Visiting an Apple Store FAQ` and `Airpods` are assigned to the expected text elements which are not visible.

Figure 6.42: Identified text is assigned to hidden text elements.

- Screenshot 6.43 contains a snippet from state 16. The sentence consists out of three expected text elements; `Get up to $235 with extra trad-in savings on Apple Watch Series 7 when you upgrade during Heart Month.`, ◊◊ and `Shop`. The OCR engine was not capable of detecting the ◊◊. The other expected text elements were identified correctly. However, our implementation assigned the text `savings` to the ◊◊. This made it impossible to completely match the expected text `Get up to $235 with extra trad-in savings on Apple Watch Series 7 when you upgrade during Heart Month.`.

Figure 6.43: OCR engine did not detect lozenge symbol correctly.

- Screenshot 6.44 shows a fragment of state 20. The expected text is `Take a look at what's new, right now.`. It is important here that the expected text contains a typographic apostrophe `'`. When we look at the screenshot, we see that the typewriter apostrophe, `'` is shown instead. The OCR engine detected the typewriter apostrophe correctly, like all other expected characters. However, our matching algorithm could not match this against the expected typographic variant.

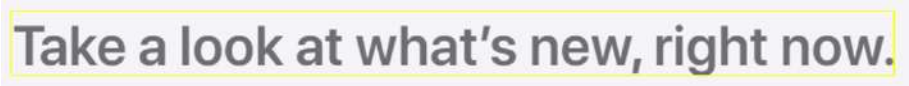


Figure 6.44: OCR engine did not detect the proper apostrophe.

Screenshot No	Expected elements	Expected elements filtered (actual)	Excluded despite visible text	Covered by action	Not detected by OCR	TPF	Entire matches	Partially matched	
								OCR	Algorithm
1	386	26(22)	0	0	16	0	5	1	0
2	386	26(22)	0	0	16	0	5	1	0
3	0	0(0)	0	0	0	0	0	0	0
4	403	32(28)	0	2	20	0	6	0	0
5	1	0(0)	0	0	0	0	0	0	0
6	896	42(38)	0	0	10	0	13	13	2
7	1	0(0)	0	0	0	0	0	0	0
8	784	33(29)	0	0	11	0	13	4	1
9	1	0(0)	0	0	0	0	0	0	0
10	813	40(26)	0	10	8	0	6	0	2
11	1	0(0)	0	0	0	0	0	0	0
12	772	30(26)	0	0	11	0	9	4	2
13	1	0(0)	0	0	0	0	0	0	0
14	603	22(19)	0	0	10	0	8	0	1
15	1	0(0)	0	0	0	0	0	0	0
16	1129	62(46)	0	0	14	0	26	3	3
17	1	1(1)	0	0	0	0	1	0	0
18	40	7(7)	0	0	0	0	5	1	1
19	0	0(0)	0	0	0	0	0	0	0
20	1129	61(46)	0	0	12	0	26	5	3
21	1	0(0)	0	0	0	0	0	0	0
22	603	22(19)	0	0	10	0	8	0	1

Table 6.7: Measurements for www.apple.com

Table 6.7 contains the measurements of www.apple.com as SUT. Our implementation has processed 7952 widgets in total. After applying our filter mechanism, 404 widgets remained as expected text elements. The filtering mechanism did not exclude any text that needed to be processed by our validation mechanism. A manual verification resulted in 329 widgets, which actually contained visible text. This means that 19% of the expected widgets should not have been validated by our implementation. 42% of the expected text has not been identified by the OCR engine and could therefore not been matched. While 39.8% of the expected text has entirely and correctly matched with the OCR results. The implementation did not find any TPF, this is according to our expectations. 4.9% could only be partially matched because of our matching algorithm. This was mainly because of OCR text elements which were assigned to surrounding text elements. 9.7% of the discovered text could only be partially matched because of a partially valid detection by the OCR engine.

6.8 Discussion

The settings configuration of TESTAR and the visual validation results from the controlled environment experiment in section 5.2 and the six target applications in section 6.1 are archived

in a GitHub [1] repository. We will now summarize the findings which we have seen while collecting the results from the six target applications. Where applicable, we will also elaborate on a possible solution.

1. With the current filtering mechanism to exclude widgets based on their ancestors, it is not possible to ignore individual elements as we have seen in section 6.2.1. The ancestor path is composed out of the elements types of the ancestors. This will lead to additional exclusions when a similar widget is placed within the same parent. By replacing this with the actual "*Path*" tag, we can improve the filtering mechanism.

2. A lot of images presented within the screenshots are identified as characters by the OCR engine, as we have seen in section 6.2.2. These OCR results affect the accuracy of the OCR engine. Because of this, the actual text is often not identified correctly. A possible solution could be to feed smaller parts of the screenshot to the OCR engine. This can be accomplished by using the location of the widgets to extract them from the screenshot and feed the extractions to the OCR engine.

3. The matcher implementation currently stores the index of the character identified by the OCR, which is last matched. Because of this, a couple of expected text elements have a lower matching percentage than expected. We have encountered this for example, in section 6.2.2. The matcher used a similar character found later in the OCR result, which leads to skipping the characters in between, which are identified by the OCR engine and which were potentially required to match the expected text. This could be solved by dropping the stored index and only try to match the unmatched OCR results.

4. We have seen that the OCR result differs across different states for the same widgets, while the widget on the screenshot is exactly the same. We have encountered this in section 6.2.2. A logical first step would be to see if we can adjust the configuration of the OCR engine to improve the detection rate. When this does not give the required improvement, we could introduce aggregating results. In order to make this work, we would first need to extract the widget from the screenshot and compare this visual with other states. For a state where we could not match the expected text entirely, we could inspect the other states. If we find a state where the expected text could be matched entirely with the OCR result and the visual of the widget is identical, we could adopt the result and mark it still as a complete match.

5. When executing a selected action, while testing a desktop application, it might happen that the execution covers a widget. When this happens, our implementation can not match the expected text because the OCR engine can not find the text, like in section 6.2.2. In order to solve this properly, we need to improve the expected text extractor. We can investigate whether the desktop protocol contains a property, which we can query for the widget which indicates whether the widget is blocked. Another approach could be that we request the *Z-index* of the widget and calculate if that is the highest value for that particular location within the SUT. As a last resort, we could mark the expected text of the widget on which the action was executed as optional. In case that the expected text cannot be matched, we can treat the result differently. For example, it can be completely ignored or annotated with a dedicated color.

6. The size and location of the screenshot is based on the main window of the SUT. We have seen that the expected text can be placed outside the boundaries of the main window. In section 6.3.2, the dialog is wider than the main window. This can be solved by iterating

[1] https://github.com/TMenting/testar-vv-data

for each state over the widgets and calculate the entire surface of the SUT. If a particular widget is placed partially or completely outside the main window, we increase the area which needs to be captured by the screenshot.

7. In section 6.3.2 we have encountered some problems with special characters which could not be matched. We need to investigate whether the OCR engine can detect them. If this is not possible, we need to introduce a dedicated ignore list for such characters. Ideally, we make this list configurable via the new settings framework.

8. Section 6.4.2 contains examples where the outline for widgets is missing. After a small investigation, it looks related to expected text, which is listed multiple times for that particular state. The log files showed that the expected text for the missing outline is matched correctly. This means that the visual border annotation is missing and that the widget is only listed once in the HTML report.

9. The matcher algorithm makes use of the "*isSpaceChar*" function. If this function returns true, we ignore the current character that we try to match. We have seen in section 6.7.2 that the SUT contains a *no-break space* character. If we feed this character to the "*isSpaceChar*" function, the result is false. The *no-break space* can not be detected by the OCR engine because it is visualized as a whitespace. Therefore, it could not be matched. In order to solve this, we need to extend the "*isSpaceChar*" check with additional characters. This can be combined with the special characters mentioned at point 7.

10. There are two types of quotes, the straight quote ' and the curly quote ’. In section 6.7.2, we have seen a screenshot which contains a curly quote. However, the OCR engine identified this as a straight quote. While they visually differ, it is acceptable to treat them similarly when it comes to matching the expected text. This can be accomplished by adding support for alternative characters for a particular character.

6.9 Threats to validity

With the usage of an OCR engine, comes the chance that a character is identified correctly by the OCR engine, while it is only partially visible. When this happens, our implementation will match it correctly with the expected text, although the text is not entirely visible. The required visible area of a cutoff character may vary between different characters in order to be identified correctly. The visual validation module includes this margin of uncertainty and can therefore not validate if the text is truly entirely visible to the user.

Our implementation can only validate one text or sentence for each GUI widget. Because of this limitation, we can not validate all the text linked to a single GUI widget. For example, for a button with a text which also has a tooltip, our implementation can only validate the presented text. In addition, it is required to trigger the tooltip before the OCR engine can retrieve the visualized text.

During the validation process of our implementation, we have noticed that the filtering mechanism to exclude certain widgets is sometime too coarse grained. The first item in section 6.8 elaborates more on this problem. Because of this problem, we have excluded more widgets than desired in order to compose a representative list of expected text widgets for validating our implementation. The influence of the reduced amount of text elements as input is therefore bigger on the overall result of the validation process.

We have seen in section 6.2.2 that some OCR results are matched with an incorrect widget in case the widget was covered. This impacts the outcome negatively. We have seen multiple scenarios were the identified text matched the expected text however, because of the incorrect location match, it could not be validated correctly. In these cases the identified text was matched with widgets which were positioned behind another widget.

In section 6.4.2, we have seen that some identified text elements are truncated by using dots. This behavior is known as ellipsize. When the text of a widget needs more space than available, the last visible characters will be replaced with dots to indicate that the text is not shown entirely. Our implementation can not identify whether a widget makes use of the ellipsize feature. Therefore, a widget which has this feature enabled will still not be marked as 100% matched.

While validating our implementation for the "*webdriver protocol*", we have seen in section 6.5.2 that we can not validate all the visible text which is shown on a website. Our implementation can not extract the expected text for embedded content. The embedded content can be anything, from an advertisement to a support chat window. A dedicated investigation is required to see whether it is possible to include the embedded content in the validation process, or that it should be validated separately.

We have encountered one case in section 6.4.2, which has been marked as 100% matched while there was no linked OCR result. This incorrect result has only been seen once. For all the other entirely matched text elements, we have a corresponding OCR result. We have tried to investigate this bug. Unfortunately, we could not find the root cause of this problem.

Chapter 7

Conclusion

7.1 Conclusion

This section will give an overview of the research questions and their answers.

RQ1: *How can OCR help to automatically validate whether the text is entirely visible?*

With the implementation of an OCR engine, we made it possible to introduce a mechanism to obtain the visualized text which is decoupled from the SUT. By feeding the OCR engine with a captured screenshot from the SUT for each new state, the engine can analyze the screenshot and identify which text is shown for that particular state. Because the OCR engine only processes the captured screenshots, it is decoupled from the SUT. The OCR engine has no knowledge about the SUT or which text should be visible. Section 4.3 elaborates on the implementation of the OCR engine. Test runs, which we executed while implementing the OCR engine, returned promising results. The OCR engine is capable of identifying the text elements located on the screenshot. With the implementation completed, we can obtain the visualized text elements in an automated way.

RQ2: *How can OCR be integrated into TESTAR in order to validate the visibility of the text for the application under test?*

In order to validate whether the text presented by the SUT is entirely visible for the user, we need to acquire the expected text and compare that against the visualized text. With the usage of an OCR engine, we can obtain the visualized text. In section 4.1.2, we have created a design for integrating the OCR engine. This design elaborates on the technical implementation and how the OCR engine is incorporated into the workflow for testing the SUT. This design also explains how we can acquire the expected text which we need in order to validate the visibility of the text shown by the SUT.

RQ2.1: *How to combine the OCR results when the expected text consists out of multiple words?*

The output of the OCR engine consists, among others, out of the identified text and the location of the identified text. We also have access to the location of the expected text. Based on the location of the expected text, we try to find related OCR results which are positioned within the location of the expected text element. We do this by iterating over all the expected text elements and for each element we find and link the OCR results which intersect with this location. In section 4.5, we explain more in detail how we combine the OCR results so we can

validate the expected text.

RQ2.2: *How can we filter out the false positives of widgets that do not have a text that needs to be visualized?*

When we query the expected text elements for each state of the SUT, we receive for all the widgets within that particular state the expected text. Unfortunately, there are widgets which do not visualize an expected text despite the fact that they do return a text when requesting it. Another scenario is that some widgets we need to query a different property in order to receive the actual text. We have solved these problems by introducing a filtering mechanism. This mechanism can exclude dedicated widgets or return the right property, which we need to query when the default property does not return the expected text. In section 4.4, we elaborate more on how we extract the expected text and how we filter the widgets which do not contain any visual text.

RQ2.3: *How to match the results from the OCR engine with the information that is extracted from the accessibility API?*

We link the OCR results and the expected text elements based on their location. This process is explained in section 4.5. After the OCR results are linked together with the expected text elements, we sort the OCR results for each expected text element based on their location. First, we calculate the line height. Based on the calculated line height, we group the OCR results for that particular expected text element. This is done based on the Y coordinate. Finally, we sort the OCR results on each line based on their X coordinate to place them in the right order. By iterating over the characters from the expected text, we try to match the sorted OCR results. For each character, we try to find the corresponding character within the linked OCR result. Because the OCR engine returns single words in our implementation, we can not detect whitespaces. Therefore, we mark whitespaces explicitly and ignore them while matching the expected text.

RQ3: *How can the implemented solution, for detecting automatically whether the text of GUI components is entirely visible, be validated?*

In order to validate that our implementation for detecting whether the expected text is shown entirely, we have defined metrics in section 5.2.2. When we run TESTAR with our implementation enabled, we can get an objective view by using these metrics. In addition to these metrics, we have tested our implementation. First, we executed an experiment in a controlled environment. For this experiment, we have modified a desktop application. We have introduced text elements which are not entirely visible for the user. After our implementation has successfully detected these modifications, we have tested our implementation against six applications. Three desktop applications and three websites. For each of these executions, we have used the metrics. The results from these tests can be found in chapter 6.

RQ Main: *How can OCR contribute to automatic validation that the text on GUI components, of an application under test by TESTAR, are entirely visible for the user?*

In order for OCR to contribute to automatic validation, whether text shown by a SUT is entirely visible, we need to first collect the expected text. In combination with our filtering mechanism, introduced in section 4.4, we can obtain the expected text for each state of the SUT. With the implementation of the OCR engine in section 4.3, we have realized an independent approach, which has no knowledge about the SUT and the text it needs to visualize, to obtain the text visualized by the SUT. The implemented matcher algorithm, in section 4.5, uses the expected text and the identified text from the OCR engine to validate whether the text of the GUI

components are entirely visible for the user. With this implementation, TESTAR becomes capable of autonomously testing the visibility of the text elements shown by the SUT.

7.2 Future work

While performing our research, we have gained new insights and were able to answer our research questions. During this period, some new questions and ideas have arisen. We have listed them below:

1. We addressed a couple of issues which we have discovered while validating our implementation in section 6.8. It will be interesting to solve these problems and running the validation of the implementation again to see whether the implementation has improved.

2. In line with the text validation, we could extend the visual validation by trying to detect more generic layout problems. Perhaps it is possible to identify layout problems, for example, by validating that the size of a widget is not bigger that the parent widget.

3. With our current implementation, we are capable of validating the text of the SUT. This process validates the text while the SUT has a particular dimension. It might be interesting to test the SUT multiple times while we change the dimensions during each run. By doing this, we can try to find the text elements which become not entirely visible. Especially when we reduce the dimensions, it is possible that text elements will not fit anymore and therefore only show their content partially.

4. The matched result of an expected text element consists of an overall percentage, which reflects the number of successful matched characters and each character of the expected text we store the individual result. We also store the OCR result, which was linked based on the location of the text elements. The overall percentage can be improved by adding a Levenshtein distance [1]. This is a string metric which can measure the difference between the two strings.

5. The current validation takes place based on the action selection algorithm. For the validation of our implementation, we have chosen to use the default random action selection. This makes it harder to validate whether all the text elements are entirely visible. This can be improved by introducing GUI coverage. If we reward the system to select an action which leads to an unexplored view of the application, we will cover faster and more text elements compared to a random selection algorithm.

6. During the literature study, we found it remarkable that a lot of researchers [2] are referring to a paper from '92 [46], where they talk about the ratio of GUI code in the source code. This paper is an extension of the original paper [47] from out of the 70s. The original paper is based on two applications, while the survey in the paper from '92, participants were asked to estimate the ratio of GUI compared to the complete source base. It has been a while that this research took place, therefore it might be interesting to redo the research with an updated questionnaire. Additionally, include open source tools in the investigation.

[1] https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

[2] https://scholar.google.com/scholar?cites=4685139532616295677&as_sdt=2005&sciodt=0,5&hl=en

Appendix A

Appendix

A.1 ExtendedSettings.xml Notepad

Listing A.1: Extended settings file for Notepad

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<root>
    <visualValidationSettings>
        <enabled>true</enabled>
        <ocrConfiguration>
            <enabled>true</enabled>
            <engine>tesseract</engine>
        </ocrConfiguration>
        <matcherConfiguration>
            <locationMatchMargin>0</locationMatchMargin>
            <loggingEnabled>false</loggingEnabled>
            <failedToMatchPercentageThreshold>75</
                failedToMatchPercentageThreshold>
        </matcherConfiguration>
        <protocol>desktop_generic/Protocol_desktop_generic</protocol>
    </visualValidationSettings>
    <tesseractSettings>
        <dataPath>C:\Users\Tycho\AppData\Local\Tesseract-OCR\tessdata</dataPath
            >
        <language>eng</language>
        <imageResolution>144</imageResolution>
        <loggingEnabled>false</loggingEnabled>
        <saveImageBufferToDisk>true</saveImageBufferToDisk>
    </tesseractSettings>
    <widgetTextConfiguration>
        <widget>
            <role>UIAScrollBar</role>
            <tag></tag>
            <ignore>true</ignore>
            <ancestor></ancestor>
        </widget>
        <widget>
            <role>UIAStatusBar</role>
            <tag></tag>
            <ignore>true</ignore>
            <ancestor></ancestor>
        </widget>
        <widget>
```

```
            <role>UIAMenuBar</role>
            <tag></tag>
            <ignore>true</ignore>
            <ancestor></ancestor>
        </widget>
        <widget>
            <role>UIAEdit</role>
            <tag>UIAValueValue</tag>
            <ignore>false</ignore>
            <ancestor></ancestor>
        </widget>
        <widget>
            <role>UIAMenuItem</role>
            <tag></tag>
            <ignore>true</ignore>
            <ancestor>::UIAMenuBar::UIATitleBar::UIAWindow::Process</ancestor>
        </widget>
        <widget>
            <role>UIAButton</role>
            <tag></tag>
            <ignore>true</ignore>
            <ancestor>::UIATitleBar::UIAWindow::Process</ancestor>
        </widget>
        <widget>
            <role>UIAButton</role>
            <tag></tag>
            <ignore>true</ignore>
            <ancestor>::UIAScrollBar::UIAEdit::UIAWindow::Process</ancestor>
        </widget>
        <widget>
            <role>WdA</role>
            <tag></tag>
            <ignore>true</ignore>
            <ancestor>::WdDIV::Process</ancestor>
        </widget>
        <loggingEnabled>false</loggingEnabled>
    </widgetTextConfiguration>
    <version>1</version>
</root>
```

A.2 ExtendedSettings.xml reference application

Listing A.2: Extended settings file for reference application

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<root>
    <tesseractSettings>
        <dataPath>C:\Users\Tycho\AppData\Local\Tesseract-OCR\tessdata</dataPath
            >
        <language>eng</language>
        <imageResolution>144</imageResolution>
        <loggingEnabled>false</loggingEnabled>
        <saveImageBufferToDisk>true</saveImageBufferToDisk>
    </tesseractSettings>
    <widgetTextConfiguration>
        <loggingEnabled>false</loggingEnabled>
        <widget>
            <role>UIAButton</role>
            <tag/>
```

```
            <ignore>true</ignore>
            <ancestor>::UIATitleBar::UIAWindow::Process</ancestor>
        </widget>
        <widget>
            <role>UIAButton</role>
            <tag/>
            <ignore>true</ignore>
            <ancestor>::UIAToolBar::UIAWindow::Process</ancestor>
        </widget>
        <widget>
            <role>UIAToolBar</role>
            <tag/>
            <ignore>true</ignore>
            <ancestor/>
        </widget>
        <widget>
            <role>UIAMenuItem</role>
            <tag/>
            <ignore>true</ignore>
            <ancestor>::UIAMenuBar::UIATitleBar::UIAWindow::UIAWindow::Process</
                ancestor>
        </widget>
        <widget>
            <role>UIAMenuItem</role>
            <tag/>
            <ignore>true</ignore>
            <ancestor>::UIAMenuBar::UIATitleBar::UIAWindow::Process</ancestor>
        </widget>
        <widget>
            <role>UIAButton</role>
            <tag/>
            <ignore>true</ignore>
            <ancestor>::UIATitleBar::UIAWindow::UIAWindow::Process</ancestor>
        </widget>
        <widget>
            <role>UIAWindow</role>
            <tag/>
            <ignore>true</ignore>
            <ancestor>::Process</ancestor>
        </widget>
    </widgetTextConfiguration>
    <visualValidationSettings>
        <enabled>true</enabled>
        <ocrConfiguration>
            <enabled>true</enabled>
            <engine>tesseract</engine>
        </ocrConfiguration>
        <matcherConfiguration>
            <locationMatchMargin>0</locationMatchMargin>
            <loggingEnabled>false</loggingEnabled>
            <failedToMatchPercentageThreshold>75</
                failedToMatchPercentageThreshold>
        </matcherConfiguration>
        <protocol>desktop_generic/Protocol_desktop_generic</protocol>
    </visualValidationSettings>
    <version>1</version>
</root>
```

A.3 Results of the controlled experiment

Below, we will investigate the visual validation outcome for each round. This includes the initial state, actions, and the state after executing the action. A legend for the screenshot annotations:

- Green is used for text elements that have been marked as a 100% match with the OCR elements.
- Yellow is for expected text elements that have a matching percentage between 100% and the configured threshold. 75% in our case.
- Orange is used for matching results below the threshold value.
- Red is used for expected text elements that could not be matched.
- Purple is an OCR result that was not matched with an expected text element.

1. Figure A.1 shows the initial state of the reference application. All the green borders in the figure indicate that our implementation correctly detected the expected text elements. However, the `Actief..` element in the right bottom corner is annotated with a yellow border. The expected text is `Actief...`, while the OCR engine only found two dots instead of one with an accuracy of 70%. The OCR text could be matched for 89% with the expected text. This was our first implemented TPF that was correctly detected by our visual validation module. The OCR engine had recognized the following text for the three purple sections around the toolbar; `07H`, `xD`, `6b`. Because of the filter, we can successfully exclude toolbar icons like the floppy disk `"H"` and scissors `"x"` from our validation process.

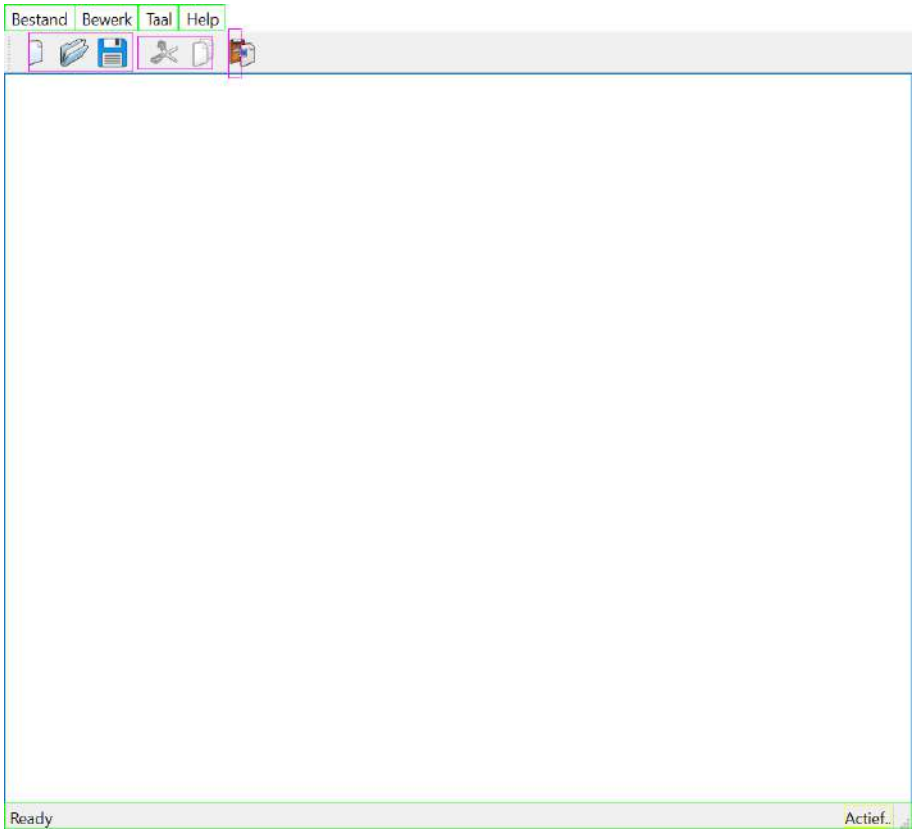


Figure A.1: Initial state of the reference application.

2. The first action is clicking on the help menu. The captured screenshot is shown in figure A.2. This has been detected correctly and the results match up with our expectations.

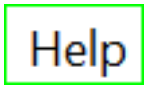


Figure A.2: The first action screenshot.

3. Figure A.3 shows the state after first action took place. The menu items are handled correctly by our implementation. The sub menu `Help` contains two items. The first entry expected, `Over`, was detected entirely. For the second entry, `Over Qt` was expected, but the OCR engine only detected `Over` with a confidence level of 91% and garbage characters `â,¬` with a confidence level of 61%. Our implementation marked this is a 71% match with the expected text. This was our second implementation of a TPF that has been detected successfully by our visual validation module. The last text element on this screenshot is the `Actief...` again. This time the OCR engine detected `Actief...` with a confidence of 54%. Because of the incorrect OCR result, the expected text has not been identified as TPF. An explanation for this incorrect detection is that the expected text element is positioned a bit left of a gray separation line that might have been identified as the last dot. To prevent this from happening, we could consider taking the confidence level into account. This incorrect detection also shows us that the OCR engine does not always provide a consistent result. Although the entire lower right area of the screenshot was the same compared to the initial state, the OCR results for the `Actief..` text element were different. The entire image affects the individual OCR results.

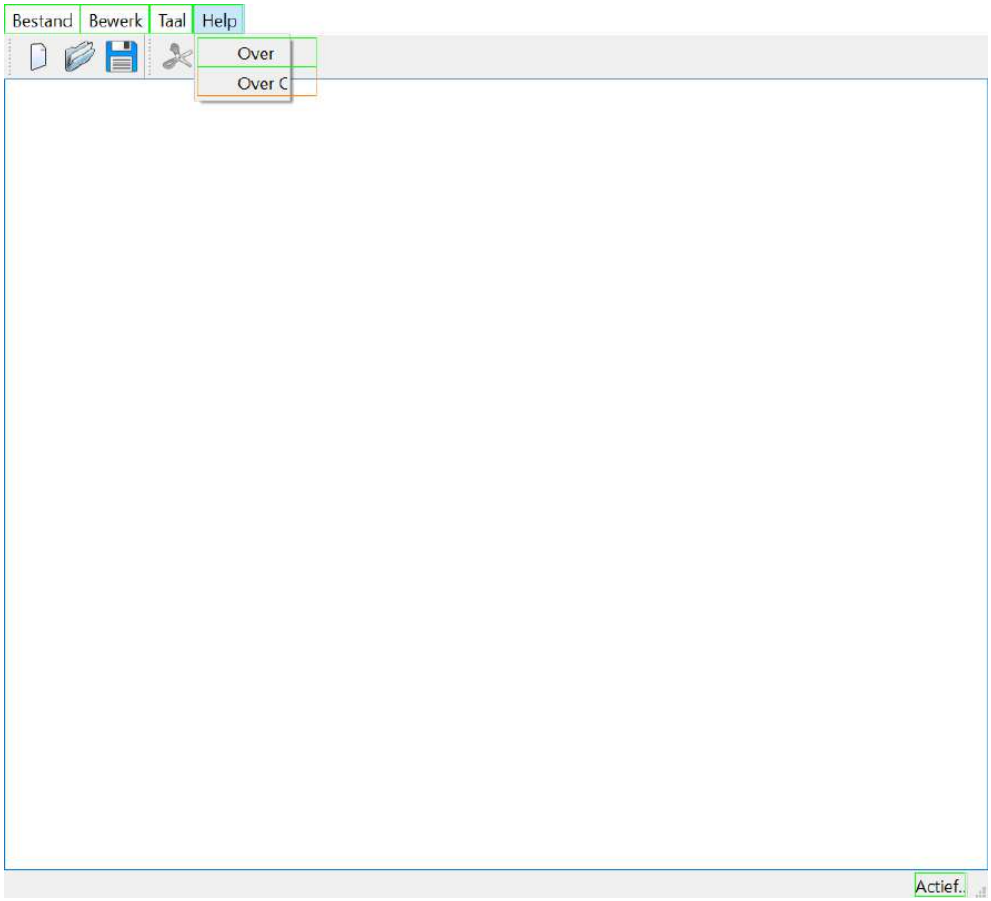


Figure A.3: The state after executing the first action.

4. The second action is clicking on the first entry of the `Help` menu. Figure A.4 shows the captured screenshot. The expected text is `Over`. Our implementation has marked this correctly as 100% matched.

Figure A.4: The second action screenshot.

5. After the second action the about dialog has appeared. Figure A.5 shows the new state. This dialog contains our third TPF. When we take a look at the results, we see that the menu items have been resolved correctly and that the `Actief...` has again been marked as fully detected, while it still misses the last dot. All the elements in the title bar of the dialog are excluded by our configuration, so the text elements detected by the OCR engine are expected to be outlined in purple. The OCR engine has detected the text element in the dialog as `Het Applicatie voorbeeld demonstreeert hoe een moderne GUI applicatie met een menu, toolbar en`. While the expected text is `Het Applicatie voorbeeld demonstreeert hoe een moderne GUI applicatie met een menu, toolbar en statusbar gemaakt moet worden met Qt.`. The visual validation module could match this for 76% and marked it yellow. Because this was our implemented TPF, this outcome was expected. The last expected text element was the `OK` located on the button of the dialog. The OCR engine detected `[oe]` with a confidence level of 42%. Our implementation has marked the first character as a case mismatch and matched the lowercase `o` with the uppercase `O`. The second character could not be matched, therefore the outcome is a 50% match. In order to resolve this, an improvement of the OCR detection is needed. While examining the results, we noticed that the expected text contains a typographical error. The text `demonstreeert` needs to be replaced by `demonstreert`. As the visual validation module currently does not perform any grammar checks, this could be a helpful feature to extend the visual validation.

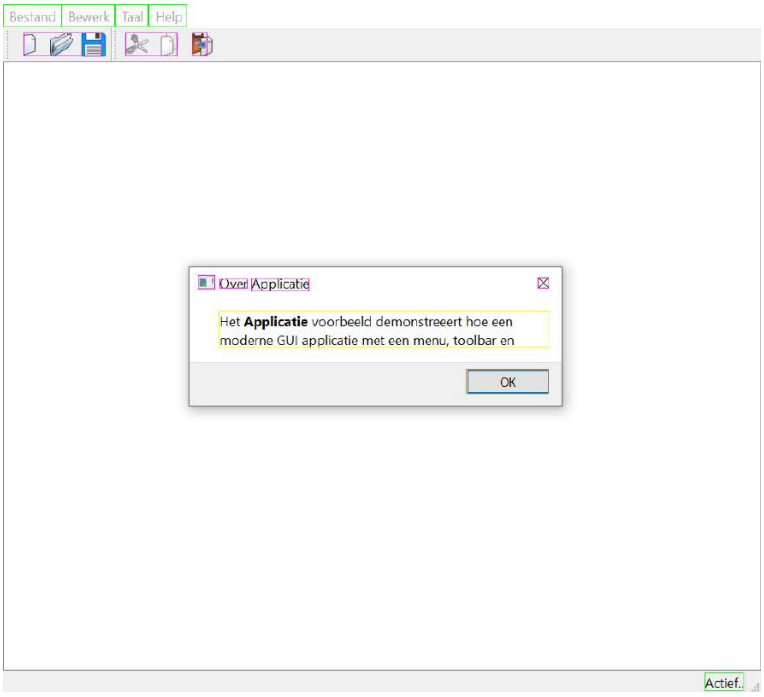


Figure A.5: The state after executing the second action.

6. The third action is clicking on the `OK` button of the dialog. Figure A.6 shows the captured screenshot. While the OCR engine could detect a few characters in the previous state, for this screenshot, not a single character has been detected. Therefore, our implementation has outlined the text in red.

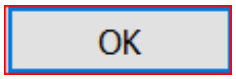


Figure A.6: The third action screenshot.

7. The state after third action is similar to the initial state, only this time the `Actief...` is again marked as a complete match. The OCR result had again a confidence level of 54%. Figure A.7 shows the new state.

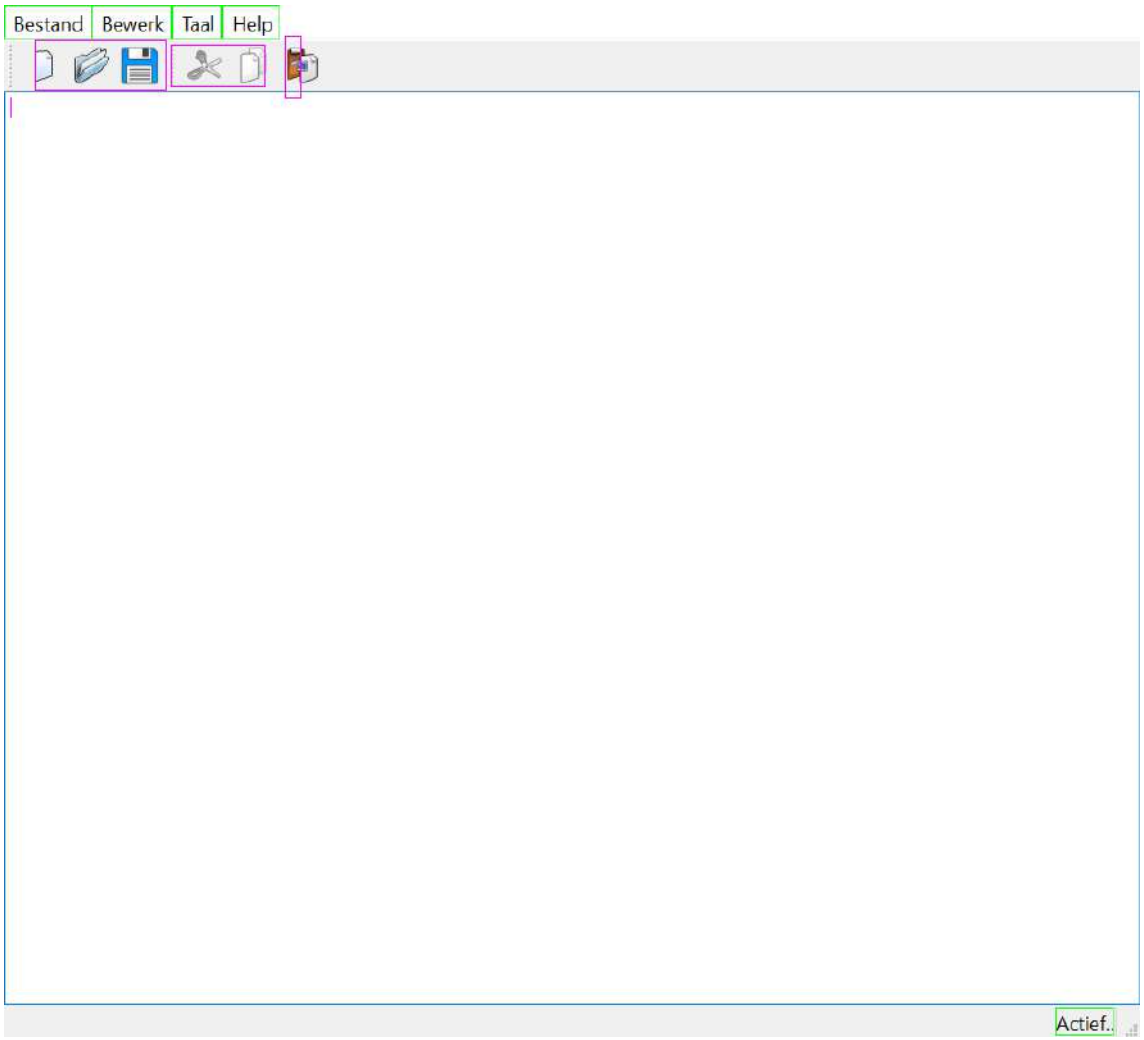


Figure A.7: The state after executing the third action.

8. The fourth action is clicking on the `Nieuw` button. Figure A.8 shows the captured screenshot. We have excluded this element from the visual validation since it does not contain any text.

Figure A.8: The fourth action screenshot.

9. After clicking on the `Nieuw` button the state is almost identical to the initial state as we can see in figure A.9. The menu text elements have been processed correctly as expected and the `Actief...` has been correctly marked. The OCR engine detected two dots, this time with a confidence level of 72%. What differs from the initial state is that the status bar contains text. The expected text is `Maak een nieuw bestand`. This has been detected entirely by the OCR engine and correctly marked as 100% matched by our implementation.

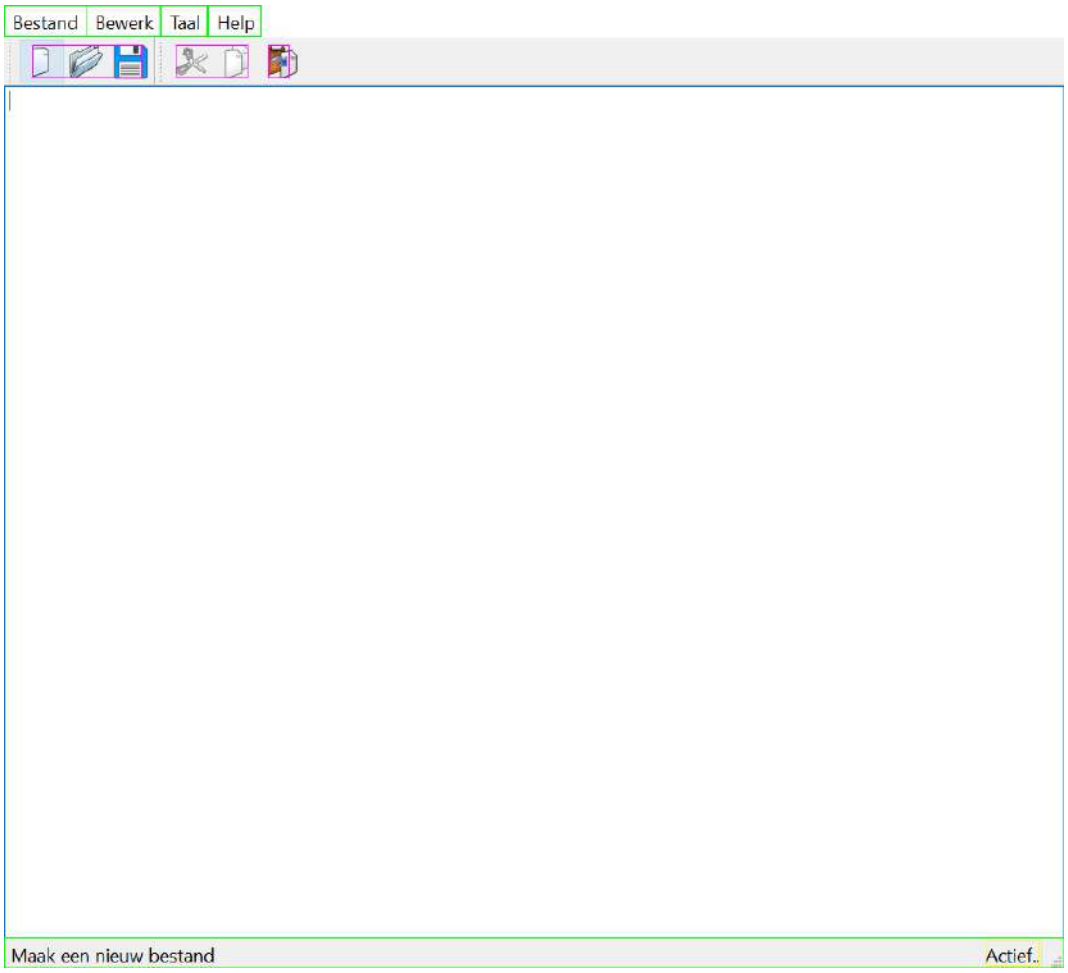


Figure A.9: The state after executing the fourth action.

10. The fifth action is again clicking on the `Help` menu. Our visual validation outcome is the same compared to the first action. Figure A.10 shows the captured screenshot.

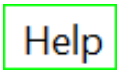


Figure A.10: The fifth action screenshot.

11. Also the state after clicking on the `Help` is identical to the second state as we can see in figure A.11. Therefore, we will skip further examination for this state.

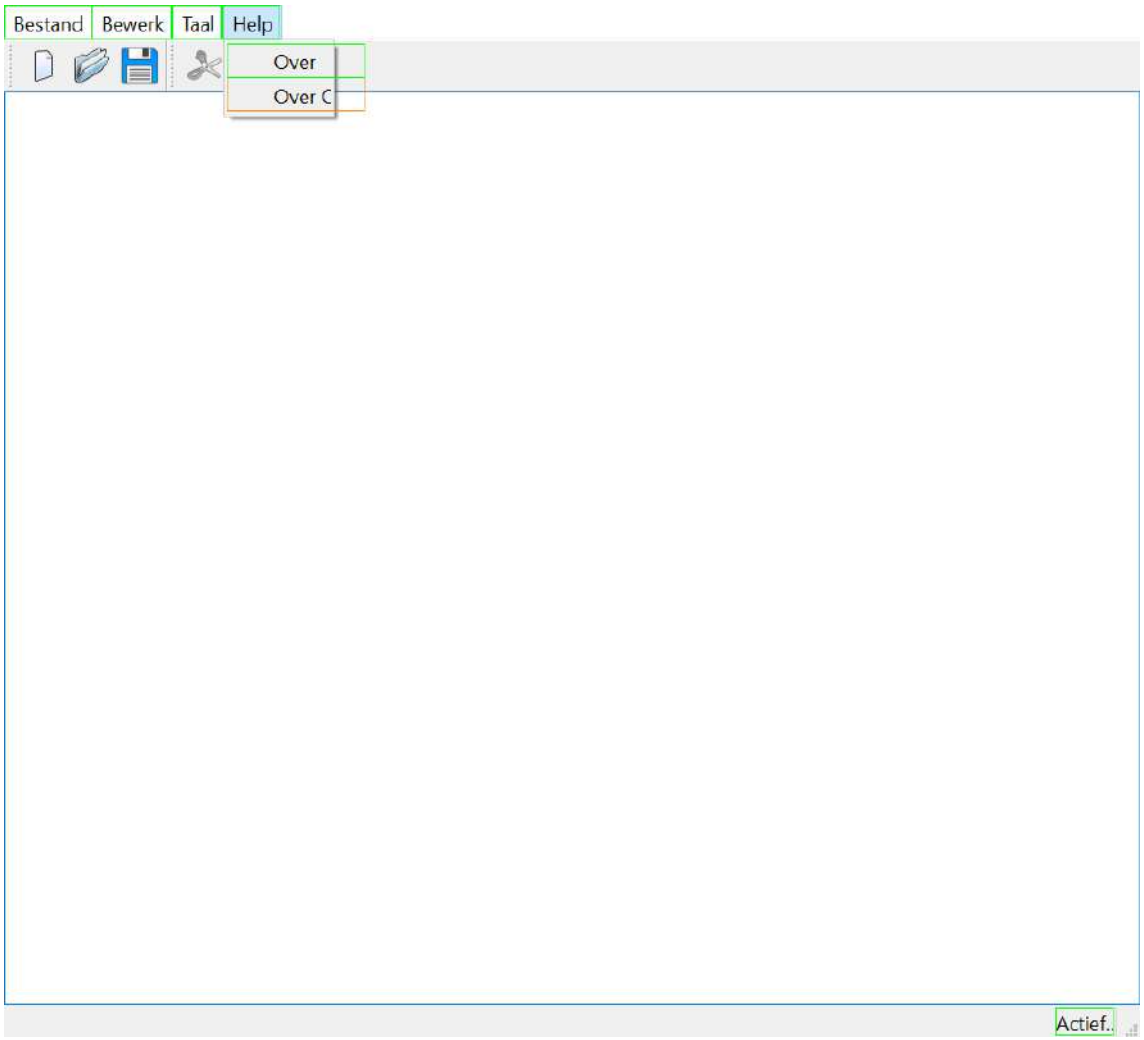


Figure A.11: The state after executing the fifth action.

12. The sixth action is identical to the second action. The `Over` is again marked correctly as 100% matched. Figure A.12 shows the captured screenshot.

Figure A.12: The sixth action screenshot.

13. The state after opening the `Over` dialog is almost identical to the state after performing the second action as we can see in figure A.13. The only difference is that the OCR engine detected `|` for the `OK` button. Therefore, the expected `OK` has a 0% match.

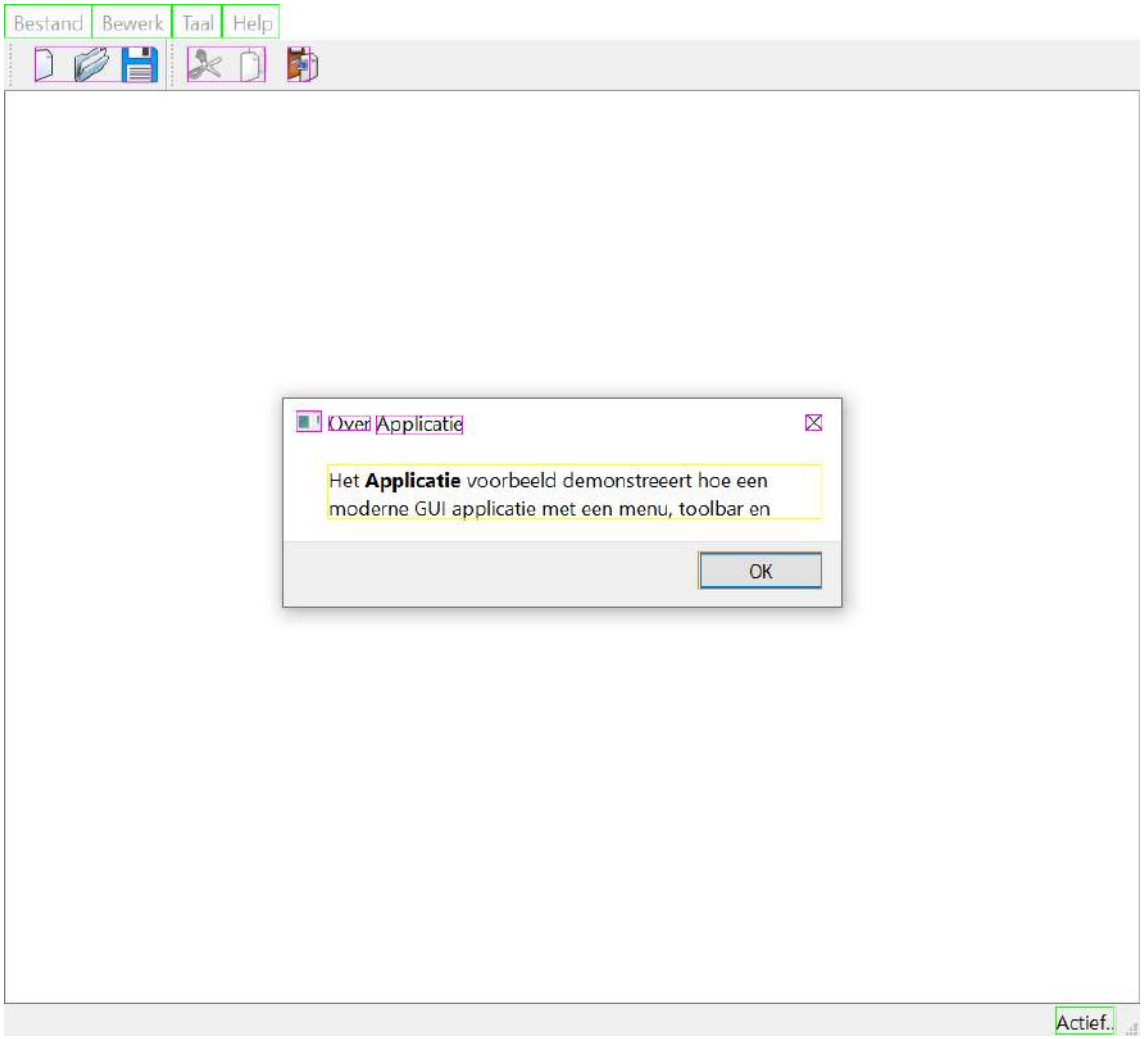


Figure A.13: The state after executing the sixth action.

14. The seventh action closes the dialog by clicking on the `OK` button. Figure A.14 shows the captured screenshot. As before, the OCR engine could not detect any characters, so our validation module outlined the element in red.

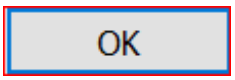


Figure A.14: The seventh action screenshot.

15. The state after closing the dialog is identical to the state after the third action as we can see in figure A.15. The visual validation outcome is the same as analyzed in item seven. Therefore, we will not analyze the results here.

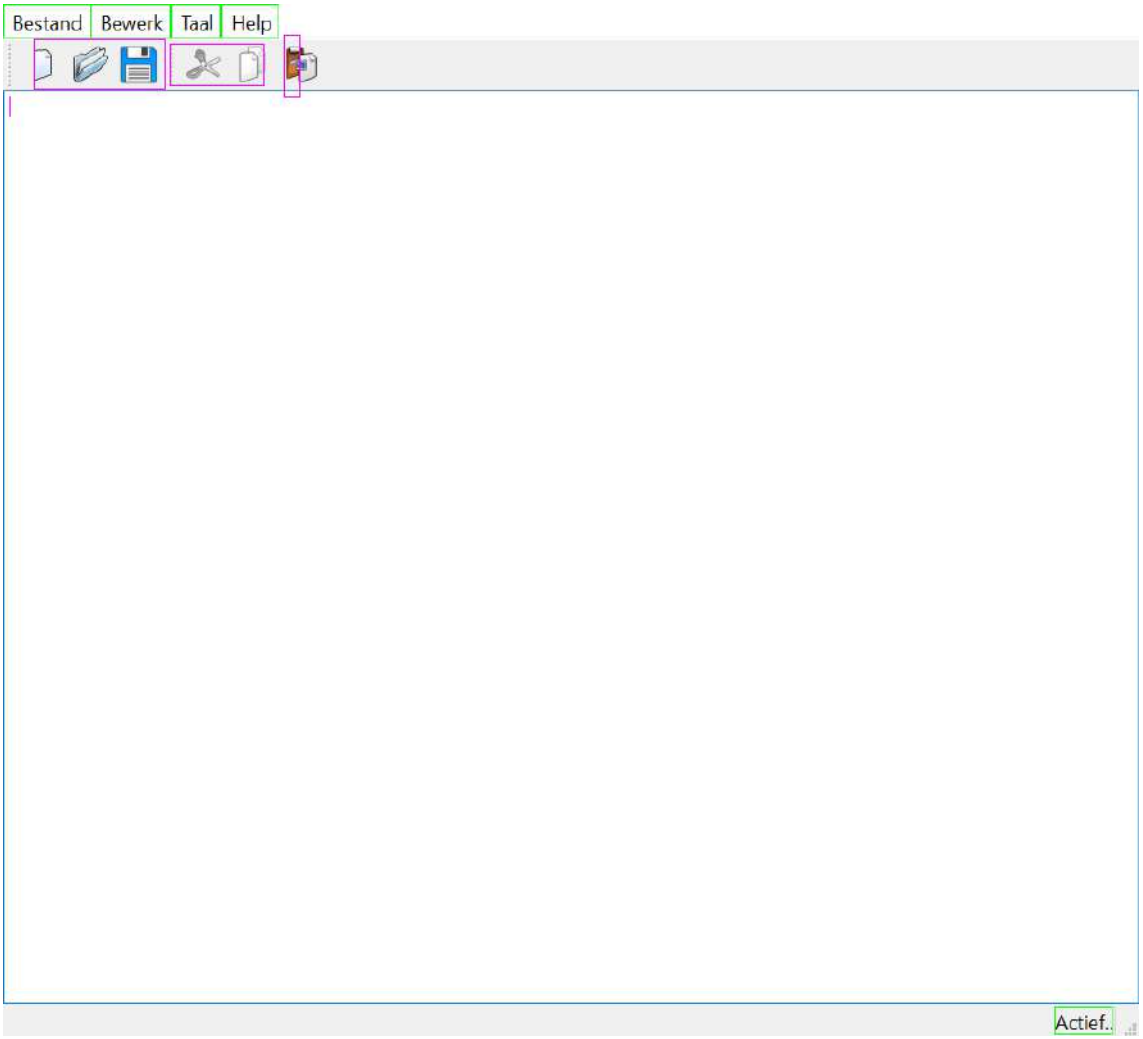


Figure A.15: The state after executing the seventh action.

16. The eight action is clicking on the `Bewerk` menu item. Our implementation calculated a 100% match, which is correct. Figure A.16 shows the captured screenshot.

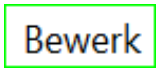


Figure A.16: The eight action screenshot.

17. After clicking on the `Bewerk` element, the menu is expanded as we can see in figure A.17. In this state, the menu items are again correctly processed by our validation mechanism. The OCR engine has detected the third dot for the `Actief...` element again with a confidence level of 54%. Because of this, the visual validation has marked this incorrect as a 100% match. The expected text elements within the `Bewerk` menu are all correctly identified and matched. We noticed that for each of the expected text elements within this menu, the shortcut actions; `Ctrl+X`, `Ctrl+C` and `Ctrl+V`, are not expected when we retrieve the expected text. These shortcuts are automatically generated by the Qt framework. While providing the translations for the reference application, the & symbol is used to identify which character should be used as a shortcut in combination with `Ctrl+`. That these shortcuts are not included when querying the expected text is because of how Qt has implemented this into their framework. Our location matching algorithm

linked the discovered text elements correctly with the expected text elements. Similar to the toolbar buttons, the OCR engine detected characters for the icons of the menu item entries.

- For the expected text `Plakken` the following OCR elements have been found:
  - `Ctrl+V` with a confidence level of 52%.
  - `Plakken` with a confidence level of 68%.
  - `(5` with a confidence level of 39%.
- For the expected text `Kopieer` the following OCR elements have been found:
  - `Ctrl+C` with a confidence level of 74%.
  - `Kopieer` with a confidence level of 87%.
  - `(D)` with a confidence level of 8%.
- For the expected text `Knippen` the following OCR elements have been found:
  - `Ctrl+X` with a confidence level of 74%.
  - `Knippen` with a confidence level of 93%.
  - `Ex` with a confidence level of 66%.

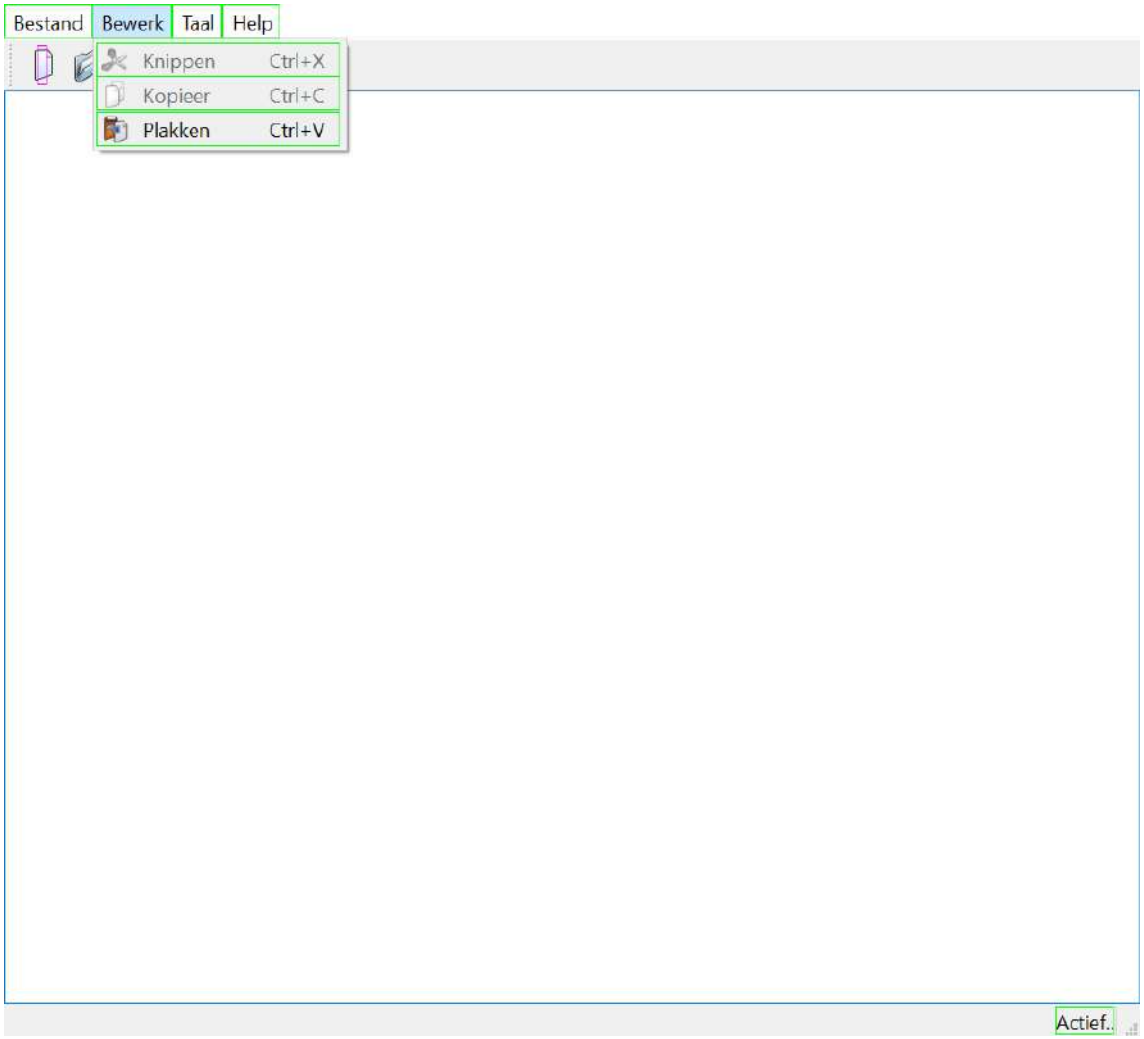


Figure A.17: The state after executing the eight action.

18. The ninth action is clicking on `Plakken`. Figure A.18 shows the captured screenshot. As explained in the previous state, the expected text does not contain the shortcut. Our implementation has marked this correctly as a 100% match.

Figure A.18: The ninth action screenshot.

19. After the `Plakken` action has been executed, the text edit contains the text that was located on the clipboard. Figure A.19 shows the new state. The text elements in the menu item are marked correctly by our implementation as a 100% match. The `Actief..` has again been marked incorrectly as a 100% match because of the third dot in the OCR result. The expected text for the text edit is `controlled environment`. The OCR engine detected two words; `controlled` with a confidence level of 96% and `environment]` with a confidence level of 20%. The visual validation module has marked the expected text element correctly as a 100% match. The OCR engine has detected the cursor in the text edit as the `]` character. Our implementation treats this character as garbage and ignores it while calculating the matching percentage.

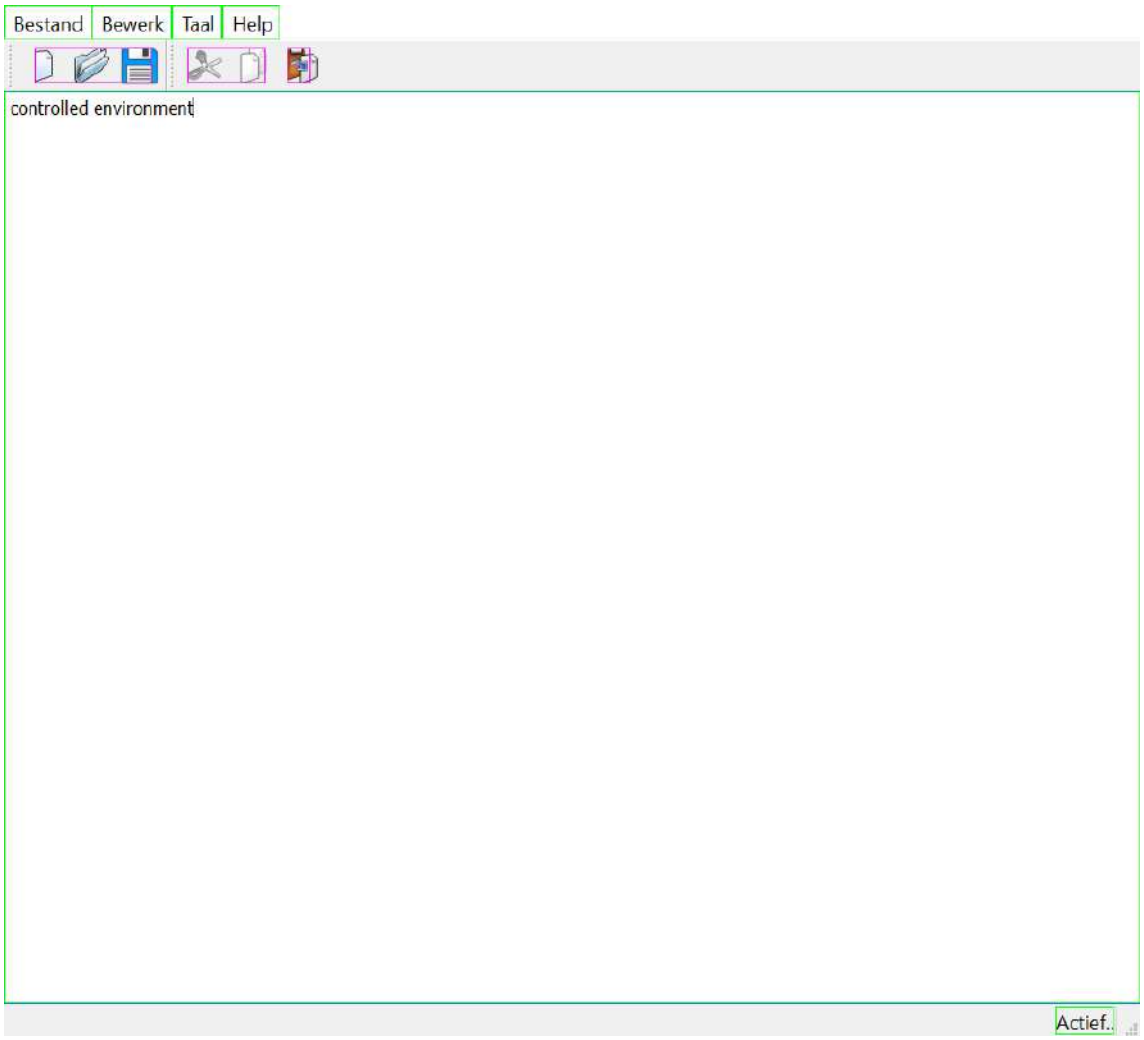


Figure A.19: The state after executing the ninth action.

20. The last action is clicking on the `Bewerk` again. The outcome is identical with the execution of the action eight as we can see in figure A.20.

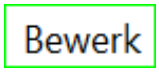


Figure A.20: The last action screenshot.

21. The validation outcome of the final state, shown by figure A.21, of the application is almost entirely correct. The only element which has been marked incorrectly is again the `Actief...` The OCR engine has detected an additional dot while it was not visualized. The menu items are identified and marked correctly as 100%. The same applies to the menu items of the `Bewerk` menu. We noticed that the OCR result of `Kopieer` had only a confidence level of 33% while previously this was 87%. However, this does not impact the matching percentage with the expected text. The last expected text element for this screenshot is the text edit. Here the expected text was `controlled environment`. The OCR engine only recognized the text `controlled`. Because of this, our implementation calculated a matching percentage of 50%. This is a correct outcome because the expanded `Bewerk` menu was shown on top of the remainder of the expected text.

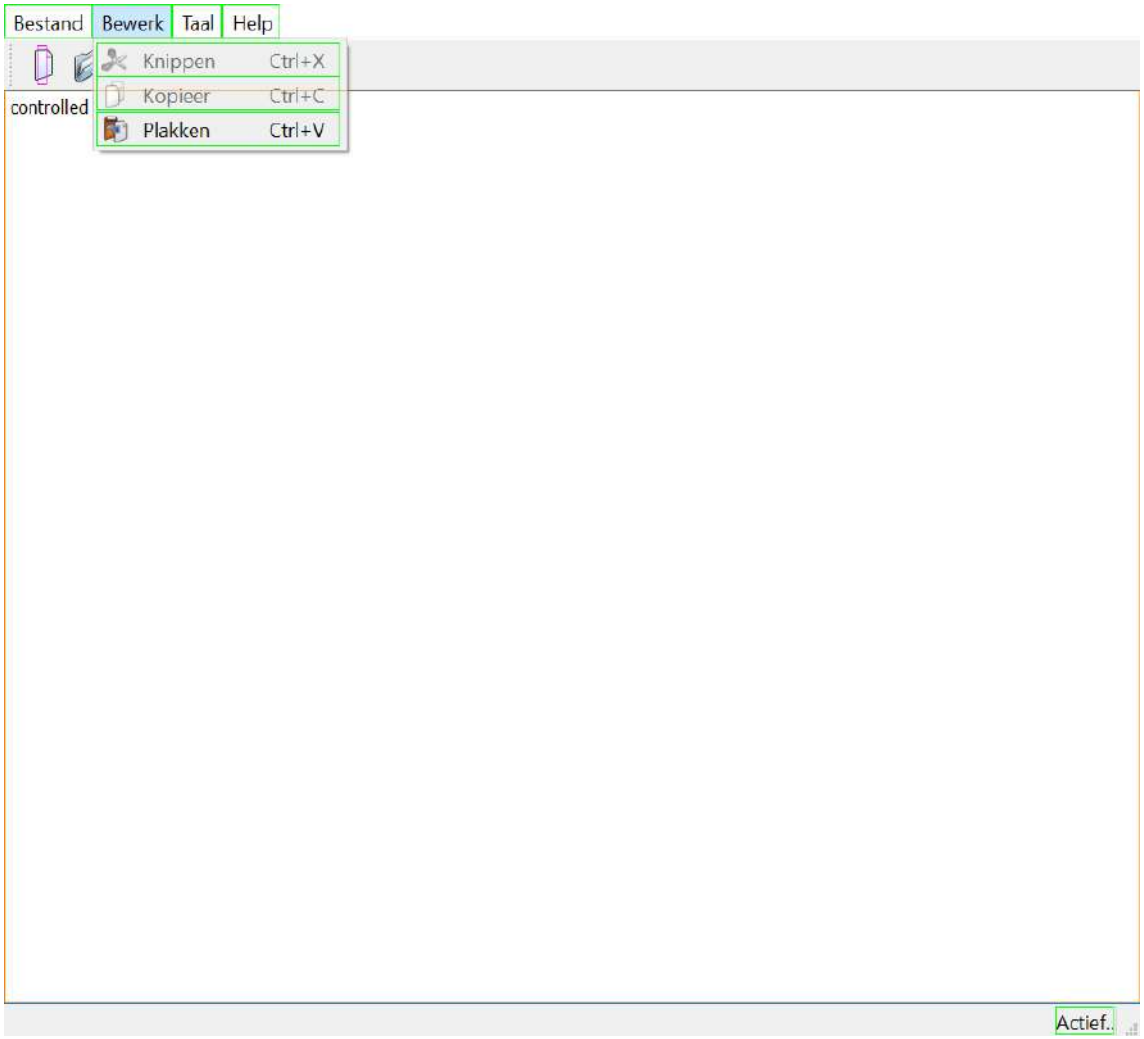


Figure A.21: The final state of the reference application.

A.4 Unfiltered expected text elements webdriver implementation result.

Result no.	Text	Location
001	'Naar content'	x=0,y=0,width=127,height=27
002	'mijnOU'	x=2047,y=0,width=126,height=55
003	'Deze website gebruikt cookies (en daarmee vergelijkbare technieken) om het bezoek voor u nog makkelijker en persoonlijker te maken. Met deze cookies kunnen wij en derde partijen uw internetgedrag binnen en buiten onze website volgen en verzamelen. Hiermee kunnen wij en derde partijen advertenties aanpassen aan uw interesses en kunt u informatie delen via social media. Klik op 'Ik ga akkoord' om cookies te accepteren en direct door te gaan naar de website of klik opom uw voorkeuren voor cookies te wijzigen. Bekijk onze voor meer informatie.'	x=364,y=1438,width=1714,height=103
004	'privacyverklaring'	x=1441,y=1506,width=133,height=18
005	'Ik ga akkoord'	x=2158,y=1474,width=157,height=27
006	'Onderwijs'	x=1428,y=54,width=201,height=150
007	'Onderzoek'	x=1630,y=54,width=211,height=150
008	'Over ons'	x=1842,y=54,width=189,height=150
009	'Contact'	x=2032,y=54,width=174,height=150
010	'en'	x=2191,y=10,width=48,height=27
011	'nl'	x=2248,y=10,width=34,height=27
012	'Direct naar'	x=549,y=6601,width=1755,height=45
013	'© 2021 Open Universiteit | | | |'	x=526,y=7144,width=1800,height=42
014	'Opleidingen'	x=549,y=1239,width=438,height=126
015	'Wetenschapsgebied'	x=988,y=1239,width=438,height=126
016	'Studeren bij de OU'	x=1426,y=1239,width=438,height=126
017	'Onderzoek'	x=1866,y=1239,width=438,height=126
018	'Disclaimer'	x=819,y=7152,width=102,height=27
019	'Privacy'	x=945,y=7152,width=69,height=27
020	'Persoonsgegevens'	x=1036,y=7152,width=183,height=27
021	'Cookiebeleid'	x=1242,y=7152,width=126,height=27
022	'Nieuws'	x=834,y=2709,width=286,height=141
023	'Alle items'	x=877,y=3805,width=198,height=94
024	'Agenda'	x=1729,y=2709,width=292,height=141
025	'Alle items'	x=1777,y=3724,width=198,height=94
026	'Bacheloropleidingen'	x=549,y=6697,width=228,height=27
027	'Masteropleidingen'	x=549,y=6747,width=210,height=27
028	'Cursussen'	x=549,y=6798,width=129,height=27
029	'Professional Programs'	x=549,y=6847,width=247,height=27
030	'Aanbod voor leraren'	x=549,y=6897,width=228,height=27

031	'Informatie over studeren'	x=549,y=6948,width=271,height=27
032	'Bestuur en organisatie'	x=1195,y=6697,width=247,height=27
033	'Promoveren'	x=1195,y=6747,width=147,height=27
034	'Samenwerken'	x=1195,y=6798,width=166,height=27
035	'Bijzondere leerstoelen'	x=1195,y=6847,width=246,height=27
036	'Werken bij de OU'	x=1195,y=6897,width=198,height=27
037	'Studiecentra'	x=1195,y=6948,width=150,height=27
038	'Service en informatie'	x=1795,y=6697,width=234,height=27
039	'Zoek medewerker'	x=1795,y=6747,width=202,height=27
040	'Persvoorlichting'	x=1795,y=6798,width=184,height=27
041	'Bezoek en postadres'	x=1795,y=6847,width=229,height=27
042	'Ict status en onderhoud'	x=1795,y=6897,width=259,height=27
043	'MijnOU'	x=1795,y=6948,width=100,height=27
044	'Giftshop Boutique'	x=1795,y=6997,width=204,height=27
045	'In de spotlight'	x=1189,y=1455,width=472,height=141
046	'11 mei 2021'	x=729,y=2926,width=652,height=30
047	'Eye tracking uit het lab naar de klas'	x=729,y=2956,width=652,height=30
048	'Informatieoverdracht in het onderwijs onderzoeken door te kijken naar oogbewegingen. Dat gebeurt niet meer alleen in het lab, maar ook steeds vaker in de klas. OU-onderzoekster Halszka Jarodzka ver...'	x=729,y=2994,width=652,height=108
049	'11 mei 2021'	x=729,y=3192,width=652,height=30
050	'Gamebrics: tijdens de gameplay feedback krijgen over je vaardigheden'	x=729,y=3222,width=652,height=60
051	'Studenten online complexe vaardigheden aanleren is niet eenvoudig - dat is tijdens de coronapandemie weer eens gebleken. Open Universiteit heeft daar al veel ervaring mee. De OU ontwikkelt games wa...'	x=729,y=3289,width=652,height=108
052	'11 mei 2021'	x=729,y=3487,width=652,height=30
053	'Strafbaarstelling van psychisch geweld: kijk het webinar terug'	x=729,y=3517,width=652,height=60
054	'Moet psychisch geweld apart strafbaar worden gesteld in het Wetboek van Strafrecht? Deze vraag stond centraal tijden het webinar van prof. mr. dr. Wilma Dreissen en prof. dr. Janine Janssen. Het we...'	x=729,y=3585,width=652,height=108
055	'12 mei 2021 16:15 – 18:00	x=1629,y=2926,width=652,height=30
056	'Online informatiemiddag Onderwijswetenschappen'	x=1629,y=2956,width=652,height=30

057	'Op woensdag 12 mei 2021 is er een online informatiemiddag van de Faculteit Onderwijswetenschappen voor studenten Onderwijswetenschappen en belangstellenden.'	x=1629,y=2994,width=652,height=81
058	'18 mei 2021 – 21 mei 2021	x=1629,y=3165,width=652,height=30
059	'Twintig OU-studenten pitchen hun idee voor digitale zorg tijdens de Dutch Health Week op 18 tot en met 21 mei'	x=1629,y=3195,width=652,height=90
060	'Van 18 t/m 21 mei 2021 vindt de eerste editie van de Dutch Health Week plaats.'	x=1629,y=3292,width=652,height=54
061	'18 mei 2021 13:00 – 14:45	x=1629,y=3436,width=652,height=30
062	'Webinar van het Gaming FieldLab 18 mei 2021'	x=1629,y=3466,width=652,height=30
063	'Twee jaar na de lancering van het DGA Gaming Fieldlab door de Open Universiteit in samenwerking met de Dutch Games Association en de RAGE-Foundation presenteren de eerste vijf Fieldlabprojecten hun...'	x=1629,y=3504,width=652,height=108
064	'Opleidingen'	x=1219,y=4036,width=415,height=141
065	'Totaal aanbod'	x=1299,y=5704,width=256,height=94
066	'•'	x=549,y=6682,width=25,height=57
067	'•'	x=549,y=6732,width=25,height=57
068	'•'	x=549,y=6781,width=25,height=57
069	'•'	x=549,y=6832,width=25,height=57
070	'•'	x=549,y=6882,width=25,height=57
071	'•'	x=549,y=6931,width=25,height=57
072	'•'	x=1195,y=6682,width=25,height=57
073	'•'	x=1195,y=6732,width=25,height=57
074	'•'	x=1195,y=6781,width=25,height=57
075	'•'	x=1195,y=6832,width=25,height=57
076	'•'	x=1195,y=6882,width=25,height=57
077	'•'	x=1195,y=6931,width=25,height=57
078	'•'	x=1795,y=6682,width=25,height=57
079	'•'	x=1795,y=6732,width=25,height=57
080	'•'	x=1795,y=6781,width=25,height=57
081	'•'	x=1795,y=6832,width=25,height=57
082	'•'	x=1795,y=6882,width=25,height=57
083	'•'	x=1795,y=6931,width=25,height=57
084	'•'	x=1795,y=6982,width=25,height=57
085	'12'	x=1449,y=2919,width=150,height=87
086	'mei'	x=1449,y=3000,width=150,height=30
087	'18'	x=1449,y=3157,width=150,height=87
088	'mei'	x=1449,y=3238,width=150,height=30
089	'18'	x=1449,y=3429,width=150,height=87
090	'mei'	x=1449,y=3510,width=150,height=30
091	'Psychologie'	x=549,y=4231,width=405,height=60

092	'Management'	x=999,y=4231,width=405,height=60
093	'Cultuur'	x=1449,y=4231,width=405,height=60
094	'Natuur en milieu'	x=1899,y=4231,width=405,height=60
095	'Onderwijs'	x=549,y=4990,width=405,height=60
096	'Informatica'	x=999,y=4990,width=405,height=60
097	'Informatiekunde'	x=1449,y=4990,width=405,height=60
098	'Rechten'	x=1899,y=4990,width=405,height=60
099	'Gezondheid'	x=549,y=5416,width=405,height=60
100	'Master Gezondheidswetenschappen'	x=571,y=1995,width=366,height=76
101	'Hoe werkt studeren bij de Open Universiteit?'	x=1020,y=1995,width=366,height=76
102	'Aanvragen van studieadvies'	x=1467,y=2026,width=366,height=45
103	'Aanmelden voor opleidingen en cursussen'	x=1914,y=1995,width=366,height=76
104	'Informatie over COVID-19'	x=571,y=2488,width=366,height=45
105	'Zo geef je online onderwijs'	x=1020,y=2488,width=366,height=45
106	'Artikelen en webcolumns'	x=1467,y=2488,width=366,height=45
107	'Onderwijsprestaties van de Open Universiteit'	x=1914,y=2457,width=366,height=76
108	'Groei. Durf. Studeer. #DenkOpen'	x=549,y=840,width=1228,height=100
109	'Nu is jouw moment'	x=549,y=940,width=1228,height=67
110	'Bachelor Psychologie'	x=549,y=4338,width=366,height=60
111	'Master Psychology - variant Arbeids- en organisatiepsychologie'	x=549,y=4399,width=366,height=120
112	'Master Psychology - variant Gezondheidspsychologie'	x=549,y=4519,width=366,height=90
113	'Master Psychology - variant Klinische Psychologie'	x=549,y=4611,width=366,height=90
114	'Master Psychology - variant Levenslooppsychologie'	x=549,y=4702,width=366,height=90
115	'Open bachelor Psychologie'	x=549,y=4792,width=366,height=60
116	'Verkorte bacheloropleiding Psychologie'	x=549,y=4854,width=366,height=90
117	'Bachelor Bedrijfskunde (BSc)'	x=999,y=4338,width=366,height=60
118	'Master Management (MSc)'	x=999,y=4399,width=366,height=60
119	'Open bachelor Bedrijfskunde'	x=999,y=4459,width=366,height=60
120	'Bachelor Algemene cultuurwetenschappen'	x=1449,y=4338,width=366,height=90
121	'Master Kunst- en cultuurwetenschappen'	x=1449,y=4429,width=366,height=90
122	'Open bachelor Algemene cultuurwetenschappen'	x=1449,y=4519,width=366,height=90
123	'Bachelor Milieunatuurwetenschappen (BSc)'	x=1899,y=4338,width=366,height=90
124	'Master Environmental Sciences (MSc)'	x=1899,y=4429,width=366,height=90
125	'Open bachelor Milieunatuurwetenschappen'	x=1899,y=4519,width=366,height=90

126	'Verkorte bacheloropleiding Milieu-natuurwetenschappen (BSc)'	x=1899,y=4611,width=366,height=120
127	'Master Onderwijswetenschappen'	x=549,y=5097,width=366,height=90
128	'Bachelor Informatica (BSc)'	x=999,y=5097,width=366,height=60
129	'Master Computer Science (MSc)'	x=999,y=5158,width=366,height=60
130	'Master Software Engineering (MSc)'	x=999,y=5218,width=366,height=90
131	'Open bachelor Informatica'	x=999,y=5310,width=366,height=60
132	'Bachelor Informatiekunde (BSc)'	x=1449,y=5097,width=366,height=60
133	'Master Business Process Management and IT (MSc)'	x=1449,y=5158,width=366,height=90
134	'Open bachelor Informatiekunde'	x=1449,y=5248,width=366,height=60
133	'Bachelor Rechtsgeleerdheid'	x=1899,y=5097,width=366,height=60
136	'HBO-rechten (ojh)'	x=1899,y=5158,width=366,height=60
137	'Master Rechtsgeleerdheid'	x=1899,y=5218,width=366,height=60
138	'Open bachelor Rechtsgeleerdheid'	x=1899,y=5280,width=366,height=90
139	'Master Gezondheidswetenschappen'	x=549,y=5523,width=366,height=90
140	'Ga ervoor'	x=549,y=1032,width=199,height=93

Table A.1: Expected text results webdriver protocol unfiltered

Acronyms

ANPR Automatic License Plate Recognition. 4

API Application Programming Interface. 3, 6, 9, 17, 18, 24, 78

CNN convolutional neural network. 8

DOM Document Object Model. 8

GUI Graphical User Interface. v, 1–3, 5–9, 21, 46, 63, 92, 95, 96

IPF Internationalization Presentation Failure. 1, 2, 8, 9, 47

OCR Optical Character Recognition. i, v, ix, 3, 4, 8, 9, 11–19, 23–29, 34, 36–43, 46, 47, 50–54, 57, 60–63, 65, 69, 71, 74, 76–79, 81, 84–96, 100–110

PF Presentation Failure. v, vii, 1–3, 8, 9

SUT System Under Test. viii, 3, 8, 9, 12, 16–18, 20–23, 26, 27, 32, 34, 41, 44–48, 50, 52, 53, 55, 57, 58, 61–63, 65, 67, 69, 71, 76, 77, 81, 82, 85–87, 90–92, 94–96

TPF Textual Presentation Failure. v, vii, viii, 2, 3, 7, 8, 10, 26, 34, 41, 44–47, 51, 52, 60, 66, 67, 69, 73, 76, 81, 86, 90, 100–102

VPN Virtual Private Network. 82

Bibliography

[1] Abdulmajeed Alameer, Sonal Mahajan, and William G. J. Halfond. "Detecting and Localizing Internationalization Presentation Failures in Web Applications". en. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Chicago, IL, USA: IEEE, Apr. 2016, pp. 202–212. ISBN: 978-1-5090-1827-7. DOI: 10.1109/ICST.2016.36. URL: http://ieeexplore.ieee.org/document/7515472/.

[2] Tommi Takala, Mika Katara, and Julian Harty. "Experiences of System-Level Model-Based GUI Testing of an Android Application". en. In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. Berlin, Germany: IEEE, Mar. 2011, pp. 377–386. ISBN: 978-1-61284-174-8. DOI: 10.1109/ICST.2011.11. URL: http://ieeexplore.ieee.org/document/5770627/.

[3] Sonal Mahajan and William G.J. Halfond. "Finding HTML presentation failures using image comparison techniques". en. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. Vasteras, Sweden: ACM Press, 2014, pp. 91–96. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642966. URL: http://dl.acm.org/citation.cfm?doid=2642937.2642966.

[4] R. Smith. "An Overview of the Tesseract OCR Engine". en. In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2*. Curitiba, Parana, Brazil: IEEE, Sept. 2007, pp. 629–633. ISBN: 978-0-7695-2822-9. DOI: 10.1109/ICDAR.2007.4376991. URL: http://ieeexplore.ieee.org/document/4376991/.

[5] Dudekula Mohammad Rafi et al. "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey". en. In: *2012 7th International Workshop on Automation of Software Test (AST)*. Zurich, Switzerland: IEEE, June 2012, pp. 36–42. ISBN: 978-1-4673-1822-8. DOI: 10.1109/IWAST.2012.6228988. URL: http://ieeexplore.ieee.org/document/6228988/.

[6] Stefan Berner, Roland Weber, and Rudolf K Keller. "Observations and Lessons Learned from Automated Testing". en. In: (2005), p. 9.

[7] Jeremy Reimer. "A History of the GUI". In: *Ars Technica* 5 (2005), pp. 1–17.

[8] Keith Bennett and Vaclav Rajlich. "Software Maintenance and Evolution: A Roadmap". en. In: (2000), p. 14.

[9] S. Yoo and M. Harman. "Regression testing minimization, selection and prioritization: a survey". en. In: *Software Testing, Verification and Reliability* (2010). ISSN: 09600833. DOI: 10.1002/stvr.430. URL: http://doi.wiley.com/10.1002/stvr.430.

[10] Ishan Banerjee et al. "Graphical user interface (GUI) testing: Systematic mapping and repository". en. In: *Information and Software Technology* 55.10 (Oct. 2013), pp. 1679–1694. ISSN: 09505849. DOI: 10.1016/j.infsof.2013.03.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584913000669.

[11] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. "Sikuli: using GUI screenshots for search and automation". en. In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology - UIST '09*. Victoria, BC, Canada: ACM Press, 2009, p. 183. ISBN: 978-1-60558-745-5. DOI: 10.1145/1622176.1622213.

[12] Tanja Vos et al. "FITTEST: A new continuous and automated testing process for future Internet applications". en. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Antwerp, Belgium: IEEE, Feb. 2014, pp. 407–410. ISBN: 978-1-4799-3752-3. DOI: 10.1109/CSMR-WCRE.2014.6747206. URL: http://ieeexplore.ieee.org/document/6747206/.

[13] W.E. Howden. "Theoretical and Empirical Studies of Program Testing". en. In: *IEEE Transactions on Software Engineering* SE-4.4 (July 1978), pp. 293–298. ISSN: 0098-5589. DOI: 10.1109/TSE.1978.231514.

[14] Yen-An Shih, Yi-Ping Chang, and Cheng-Zen Yang. "An Automated Detection Framework for Testing Visual GUI Layouts of Android Applications". en. In: *Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering*. WCSE, 2017. ISBN: 978-981-11-3671-9. DOI: 10.18178/wcse.2017.06.093. URL: http://www.wcse.org/WCSE_2017/093.pdf.

[15] Cheng-Zen Yang et al. "LAD: A Layout Anomaly Detector for Android Applications". en. In: July 2019, pp. 557–562. DOI: 10.18293/SEKE2019-186. URL: http://ksiresearchorg.ipage.com/seke/seke19paper/seke19paper_186.pdf.

[16] Yaohui Wang and Fudan University. "Textout: Detecting Text-Layout Bugs in Mobile Apps via Visualization-Oriented Learning". en. In: (2019), p. 11.

[17] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. "Automated layout failure detection for responsive web pages without an explicit oracle". en. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017*. ACM Press, 2017, pp. 192–202. ISBN: 978-1-4503-5076-1. DOI: 10.1145/3092703.3092712. URL: http://dl.acm.org/citation.cfm?doid=3092703.3092712.

[18] Juichi Takahashi. "An Automated Oracle for Verifying GUI Objects". en. In: *ACM SIGSOFT* 26.4 (2003), p. 6.

[19] Rudolf Ramler, Thomas Wetzlmaier, and Robert Hoschek. "GUI scalability issues of windows desktop applications and how to find them". en. In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. Amsterdam Netherlands: ACM, July 2018, pp. 63–67. ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236491. URL: https://dl.acm.org/doi/10.1145/3236454.3236491 (visited on 04/24/2022).

[20] Rudolf Ramler and Robert Hoschek. "Process and Tool Support for Internationalization and Localization Testing in Software Product Development". In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2017, pp. 385–393.

[21] *Adobe Acrobat*. URL: https://helpx.adobe.com/document-cloud/help/using-ocr-exportpdf.html.

[22] *ABBYY Cloud OCR SDK*. URL: https://www.abbyy.com/cloud-ocr-sdk/features.

[23] *ABBYY FineReader Engine*. URL: https://www.abbyy.com/ocr-sdk/technical-specifications.

[24] *Amazon Textract*. URL: https://aws.amazon.com/textract.

[25] *Asprise OCR*. URL: http://asprise.com/royalty-free-library/ocr-api-for-java-csharp-vb.net.html.

[26] *Calamari*. URL: https://github.com/Calamari-OCR/calamari.

[27] *CuneiForm*. URL: https://launchpad.net/cuneiform-linux.

[28] *Dynamsoft*. URL: https://www.dynamsoft.com/Products/.net-ocr-component.aspx.

[29] *GOCR*. URL: https://www-e.ovgu.de/jschulen/ocr.

[30] *Google Cloud*. URL: https://cloud.google.com/vision/docs/ocr.

[31] *Kraken*. URL: http://kraken.re.

[32] *Azure's Computer Vision API*. URL: https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/concept-recognizing-text.

[33] *OCRvision*. URL: https://www.ocrvision.com.

[34] *OmniPage*. URL: https://www.kofax.com/Products/omnipage.

[35] *OpenText*. URL: https://www.opentext.com/products-and-solutions/partners-and-alliances/opentext-oem-solutions/advanced-recognition-technologies.

[36] *Ocrad*. URL: https://www.gnu.org/software/ocrad.

[37] *Ocropy*. URL: https://github.com/ocropus/ocropy.

[38] *Ocular*. URL: https://github.com/tberg12/ocular.

[39] *Readiris*. URL: https://www.irislink.com/EN-NL/c1897/IRIS-OCR-SDK.aspx.

[40] *Rossum*. URL: https://rossum.ai/product.

[41] *SimpleOCR*. URL: https://www.simpleocr.com/ocr-sdk.

[42] *Transym*. URL: http://transym.com/index.htm.

[43] *Microsoft UWP OCR*. URL: https://blogs.windows.com/windowsdeveloper/2016/02/08/optical-character-recognition-ocr-for-windows-10.

[44] *Tegaki*. URL: https://tegaki.github.io.

[45] *Tesseract*. URL: https://github.com/tesseract-ocr.

[46] Brad A. Myers and Mary Beth Rosson. "Survey on user interface programming". en. In: *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '92*. Monterey, California, United States: ACM Press, 1992, pp. 195–202. ISBN: 978-0-89791-513-7. DOI: 10.1145/142750.142789. URL: http://portal.acm.org/citation.cfm?doid=142750.142789.

[47] Jimmy A Sutton and Ralph H Sprague. *A study of display generation and management in interactive business applications*. IBM Thomas J. Watson Research Division, 1978.