Automated GUI Ripping for Web Applications

Domínguez Osorio, Juan Manuel
jm.dominguez@uniandes.edu.co
López Fabara, Andrés Felipe
ap.lopezf@uniandes.edu.co
Advisor: PhD. Linares Vásquez, Mario
Universidad de los Andes

A thesis submitted for the degree of Systems and Computing Engineering

Bogotá, May 2019
This thesis is dedicated to...

*Everyone who fights for their dreams.*
Acknowledgements

Firstly, we will like to thank our families as they have given us their support throughout our career. They have cheered us when we needed it and have given us strength in the most difficult moments. We also would like to thank our friends, as they have enabled us to grow as persons and have given us numerous moments of joy. Another big thank you to the teachers that we had during our time in the school and university. The knowledge that we have acquired during this years is invaluable. Finally, thanks to Mario Linares, our adviser, and the testing investigation team of The Software Design Lab, as they played a crucial role during the development of this project.
INDEX

RESUMEN ................................................................. 4
ABSTRACT ................................................................. 5
INTRODUCTION ......................................................... 6

I  Prior work ......................................................... 11
1.1 GUI ripping theory .............................................. 11
   1.1.1 GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing ........................................... 11
1.2 Platform specific projects ...................................... 12
   1.2.1 AndroidRipper ............................................. 13
   1.2.2 GUITAR .................................................. 13
   1.2.3 Testar ................................................... 14
   1.2.4 RIP .................................................... 14
   1.2.5 SlumDroid .............................................. 15

II  Proposed approach ............................................... 16
2.1 Design of the testing tool ...................................... 16
   2.1.1 Puppeteer .............................................. 16
   2.1.2 Exploration algorithm .................................. 19
   2.1.3 GUI state .............................................. 20
   2.1.4 Oracle definition ...................................... 21
   2.1.5 Model definition ...................................... 22
   2.1.6 Report design ........................................ 22
2.2 Implementation ................................................ 23
   2.2.1 Selection of front-end frameworks ..................... 23
   2.2.2 GUI exploration and interaction ....................... 24
2.3 Evaluation ...................................................... 30
   2.3.1 Comparison with Cypress ............................. 31
   2.3.2 Identifying and correcting mistakes .................. 32
List of figures

2.1 Client side rendering response in React ........................................... 17
2.2 Configuration file example ............................................................. 27
2.3 Graph example .............................................................................. 29
2.4 State example ................................................................................ 30
RESUMEN

En los últimos años, las pruebas de software se han convertido en un factor clave para entregar productos de calidad a los clientes. Una de las técnicas que ha tomado fuerza en los últimos años es GUI ripping. En este tiempo, se han desarrollado diferentes herramientas para realizar ripping en diferentes plataformas. Sin embargo, la evolución continua de diferentes tecnologías como los frameworks para desarrollo de front-end representan un desafío para estas herramientas. Por este motivo, en el presente documento, presentamos una herramienta extendible de GUI ripping, que es compatible con dichos frameworks.
ABSTRACT

In later years testing has become a key factor for delivering quality software to the end user. One of the techniques that has grown during this years is GUI ripping. Many GUI ripping tools have been developed for different platforms, however, with the continuous evolution of technologies such as web frameworks a challenge has risen to develop tools that are compatible with modern technologies. In this paper we present a extendible GUI ripping tool that is compatible with modern front-end web frameworks.
INTRODUCTION

Web applications are a common thing nowadays and it is normal for a user to visit multiple web pages in a single day from the thousands of options available on the web. The presentation layer of each one of this web applications is their graphical user interface (GUI) and it is a crucial element in its operation because it is where the users interact with the application in order to access the different functionalities. Taking this into account, the GUI plays an indispensable role when it comes to a web applications success and acceptance. However, there are also other factors that need to be taken into account when evaluating a web application, such as the presence of software bugs. A software bug is defined as a failure built in the system that causes it to produce an unexpected result or malfunction. In web applications, there are several software bugs that users deal with, as broken links, incorrect arithmetic operations, or null pointer exceptions. These errors lead to an unsatisfying experience and therefore to user rejection.

On another note, testing is a methodology which focuses on determining the quality level of a software system. There are different approaches, levels and types of testing, however, this project will focus on GUI Ripping. It is a testing technique that is used to evaluate robustness of web applications. On a high level, a ripper identifies the elements of the GUI that the user could interact with. Then it generates a series of events (clicks, swipes, form fills) and monitors the state changes on the system under testing, until the application has been explored with a predefined search strategy. The objective of a ripper is to systematically explore the GUI in
order to identify software bugs and report them to testers and developers.

In addition, headless testing is a technique with the purpose of reducing the consumption of resources during the testing process. This is achieved by using an environment without a GUI, it means, by testing the application without seeing its interface. This technique implies faster and less greedy tests. A consequence of headless testing is the reduction of resources that are spent in the testing process of an application, which allows for more resources being destined to other activities, like application development.

Finally, in recent years a wide variety of front-end web frameworks such as React\(^1\), Angular\(^2\) and Vue\(^3\) have been developed. However, it is a challenge for GUI ripping as the different frameworks implement different techniques to implement client based rendering. For example, React is a component based framework, in which each component manages its own state; it also uses the React DOM to update elements, as opposed to the browser DOM. The different rendering techniques used by these frameworks can have negative effects on GUI rendering. For instance, when capturing links via JQuery selectors in React, some empty urls may be obtained.

Taking all of these factors into account, the main idea of this project is to develop an open source and extensible GUI ripping testing tool. This testing tool must allow for headless (for resource consumption purposes) and headful (for debugging purposes) testing. Additionally, it must be able to function in at least two modern front-end development frameworks. This tool could help improve the testing of web applications, both in resource expendage and coverage.

\(^1\)https://github.com/facebook/react
\(^2\)https://angular.io/
\(^3\)https://vuejs.org/
Project goals

Main goal

Our main goal is to develop an open source extensible GUI Ripping testing tool for web applications.

Specific goals

1. Design the testing tool, including its models, the reports that are going to be generated and all the components required for the GUI ripping process. All of this based on prior work.

2. Implement the designed GUI ripping tool using Google’s Puppeteer; this testing tool should work in at least two modern front-end frameworks.

3. Evaluate the created solution by comparing it with state of art tools like Cypress.

Methodology

The development of the testing tool was divided in three phases. The first phase consisted of a bibliographic review on GUI ripping and the definition of the elements that would compose the testing tool, this elements included the model, the snapshots, the memory, the exploration heuristic and the reports that the test would subsequently generate. On a high level, this stage mainly focused on defining how GUI states would be defined, (as several approaches were evaluated), how we would explore them, what would be identified as an error and how the errors identified would be presented to the final user.
In addition, in this stage the technologies (libraries and frameworks) that were considered useful for the purpose of the project were defined, to some extent. The main decision, made was that the testing tool would be developed on top of Puppeteer\textsuperscript{4} it is a Node library that is used to control headless Chrome or Chromium (it also allows head full execution). Taking advantage of the headless features, Puppeteer is suitable for developing automated tests in those browsers. It provides different functionalities, for instance, taking screenshots of the GUI, automating form submission and executing applications on different viewports. It is worth noting that Puppeteer is maintained by Google as an open source project.

Then, the second phase consisted in the implementation of the testing tool, following the designs and technologies defined in the previous stage. This was the longest phase of the project because it included the learning curve and the time required for getting familiar with the technologies used in the development. Additionally, the second phase was composed by the definition of the frameworks that the testing tool would support. The decision process included the research of the available front-end frameworks and figuring out how the characteristics of each one of them could affect the implementation of the GUI ripping tool.

Finally, the last stage of the project was divided in substages, the first one included an evaluation process, where the developed tool was compared against state of art tools like Cypress\textsuperscript{5}. Additionally, the testing tool was executed on open source web pages such as Hábitica and Wordpress, to test coverage and functionality. On the other hand, the second stage of the project was a correction and improvement process, considering the elements identified in the evaluation substage.

\textsuperscript{4}https://github.com/GoogleChrome/puppeteer
\textsuperscript{5}https://www.cypress.io/
**Expected results**

At the end of the project, there should be a clear design for the GUI ripping tool. This design defines every component of the proposed solution and it is the base of the implementation process. Taking this into account, the GUI ripping tool should be implemented accordingly. This testing tool must be able to explore a web application and test its graphical user interface. Also, it should generate reports based on the defined model and the exploration done to the application.

Additionally, the created solution should be evaluated in order to get an idea of its quality. The evaluation will consist in comparing our implementation with similar state of art tools. Based on the results of the evaluation, the created solution will be improved by correcting the issues that were identified.

**Obtained results**

A GUI ripping tool called RIPuppet was developed. Built on top of Google Puppeteer, it is a node library that uses a DFS exploration algorithm to obtain models of web applications. In addition it identifies errors by reading output from the browser JavaScript console. The library built also generates a testing report in which a interactive visualization is shown. The visualization is a graph that is constructed using d3.js, in said graph the nodes are GUI states discovered during exploration and the edges are user interactions that generate transitions between states. For each state, the report also shows a GUI screenshot, the url, the number of errors and the number of warnings found. For each edge, the report shows the type of interaction and a screenshot of the widget.
Chapter I

Prior work

A plethora of work has been done on the field of GUI ripping. Some of it covers general GUI ripping theory, while other focus on specific platforms such as Android, iOS and web applications. In this section, some of the projects and investigations that have been done will be presented.

1.1 GUI ripping theory

1.1.1 GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing

GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing [1], was developed with the goal of specifying GUI Ripping as a dynamic process, in which a test model is engineered by systematically exploring the GUI (by opening all the windows and extracting the widgets). It defines different concepts that are useful to the implementation of this project, such as what elements form a GUI and a GUI forest, and how to reverse engineer the GUI model.
by using a DFS recursive algorithm. In addition, they identify two problems that must be taken into account for the implementation of this project.

- **Infeasible windows:** It is mentioned that the proposed approach prevents fully automation, because there are infeasible paths. An example that is given is that some windows might not be available until a valid password is provided. Taking this into account, a feature to improve window coverage (by allowing the ripper to type valid inputs) should be implemented.

- **The definition of GUI:** A GUI is defined as a front end of a software system. It accepts user input and contains widgets. Also, each widget has a fixed set of properties with discrete values. However, it is said that this definition should be extended to take into account timing constrains between objects, video players and non deterministic GUIs. Taking into consideration that the purpose of this project is to develop a ripping tool for modern web applications, the definition of GUI should be extended.

### 1.2 Platform specific projects

These are projects that implement GUI ripping techniques in specific platforms. These projects include AndroidRipper [2], GUITAR [3], Testar[^1^], Rip[^2^], SlumDroid [4] and CrashScope [5]. Most of them are tools for GUI ripping on Android applications, however there are some elements that can be considered relevant for the purpose of this project.

[^1^]: https://testar.org/
[^2^]: https://github.com/TheSoftwareDesignLab/rip
1.2.1 AndroidRipper

AndroidRipper \[2\] is an automated testing technique that is used to test Android applications by interacting with the GUI. It executes tasks (defined as a couple of an action and a GUI state) from a task list, where the starting tasks are the ones that are executable from the initial GUI of the application. While AndroidRipper explores the application, the task list is updated with the tasks of the discovered GUI states. Also, AndroidRipper constructs a GUI tree with GUI states and transitions during its execution. On another note, the work on AndroidRipper introduces the concept of GUI exploration criterion as a logical predicate that indicates if in a determined GUI state, the exploration should continue or should be terminated. A given example is the case in which a state that has already been visited is reached.

1.2.2 GUITAR

GUITAR \[3\] is a testing framework for GUI based applications. It consists of four major components: a ripper, a graph converter, a test case generator and a replayer. Firstly, the ripper generates a model of the system under test, by systematically exploring the GUI. Secondly, the graph converter changes the model into a graph. Thirdly, the test case generator utilizes the graph to perform automated test case generation. Finally, the replayer executes and evaluates the tests cases that were generated on the previous step. GUITAR is easy to extend as it supports plugins, and the approach that it implements can be applied to different types of applications: web, desktop and or mobile applications. Platforms that currently are supported by GUITAR include Java JFC, web, Java SWT, Android, iOS and UNO. Still, major adjustments may be needed to make the tool compatible with modern web frameworks. The tool is currently hosted in SourceForge\[3\]. However it remains unmaintained since 2016.

\[3\]https://sourceforge.net/projects/guitar/
1.2.3 Testar

Testar is a GUI testing tool that was developed under the FITTEST [6] project. It is an automated testing tool that executes automated tests at UI level. The structure of the UI is derived through the accessibility app of the underlying operating system. Initially, Testar generated random inputs to conduct test cases, however in order to be able to explore complex GUIs it has added more refined techniques such as prolog specifications and machine learning algorithms like Q-learning [7]. Testar can be used with desktop, web and mobile applications according to the official web page [4]. Nonetheless, the compatibility features are not quite clear, as the only documentation about this reads that the SUT must "respect the accessibility API of the underlying Operating System".

1.2.4 RIP

RIP [5] is a Java testing tool developed to enable GUI ripping on Android applications. It can be used both in emulators and physical devices. In addition, it is compatible with native and cross-platform applications. Rip implements a multi-model approach to test applications [8]. A multi-model is defined as the combination of the GUI model (the elements in the app GUI), the context model (the surrounding conditions of the app such as network connection), and the usage model (how users interact with the application). This means that during execution RIP simulates user interaction while it collects information of the GUI, the sensors and the resources of the device.

The architecture of RIP is described as follows:

1. GUI analyzer: It extracts the GUI hierarchy model of the application under test. It dif-

---

4https://testar.org/about/
5https://github.com/TheSoftwareDesignLab/rip
differentiates each state by comparing the activity name, and the XML of each discovered state.

2. Input analyzes: It detects input components and generates random input depending on the component type.

3. Sensor analyzer: It detects which sensors can be accessed from the application using the android manifest. Additionally, it can detect the state of each sensor via ADB commands.

4. Connectivity analyzer: It can detect the connectivity status of the device during the GUI ripping process. It can also trigger connectivity changes.

5. Static analyzer and GATOR: Builds a graph by doing static analysis. This is done through the GATOR tool.

6. RIP GUI: It interacts with ADB to explore the application, and coordinates the execution of the other RIP components.

1.2.5 SlumDroid

Slumdroid [4] is a modified version of AndroidRipper [2] that generates a model of the application under test and analyzes said model, to implement input perturbation techniques and consequently, generate perturbed inputs that can be used in later test sessions. The input perturbation operations include removing the mandatory sets, disordering the sequence of sets and inserting invalid or dangerous characters.
Chapter II

Proposed approach

This chapter explains the steps that were followed to fulfill the main goal of the project. Each one of the sections coincide with the phases that were defined in the methodology section.

2.1 Design of the testing tool

The name chosen for the testing tool was RIPuppet. It was developed on Javascript on top of Puppeteer. Puppeteer was selected before starting the implementation for various reasons.

2.1.1 Puppeteer

Puppeteer [9] is a NodeJS library which provides a high level API to control Chrome or Chromium over the browser DevTools protocol [10]. It is a low level protocol that allows developers to debug, instrument, profile and inspect Chromium and Chrome. Puppeteer was selected to implement the testing tool because of the complete set of functionalities that it offers, as it allows to take screenshots of the visited webpages, provides methods to automate form submissions, and
Figure 2.1: Client side rendering response in React

```html
<DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <link rel="shortcut icon" href="/favicon.ico">
  <title>React App</title>
</head>
<body>
  <div id="root"></div>
  <script src="/app.js"></script>
</body>
</html>
```

supports user like GUI interaction (like clicks, keyboard inputs and scroll actions). In addition, Puppeteer supports methods for Server Side Rendering (SSR), which is necessary to execute GUI ripping on modern web frameworks such as React.

In modern web frameworks, web application content is rendered on the client via executing JavaScript scripts. SSR is a technique in which the JavaScript code is executed on the server, thus returning a static HTML file with all the content already rendered. Said technique is usually used to improve performance on some web pages, however, it is key for GUI ripping on modern web applications because it enables the GUI ripping algorithm.

According to Memon, et al. [1] one of the steps of the recursive DFS algorithm to explore GUIs and generate the GUI tree, is to get the widget list and identify the executable widgets. In web applications this could be achieved by using JQuery selectors. Nonetheless, this approximation would not work on modern web frameworks like React, as the HTML that is returned by the server has an empty div that references the app.js were the JavaScript code is hosted (This can be observed in figure 2.1). This problem is solved by implementing SSR with Puppeteer.

Furthermore, Puppeteer also provides a method to await till the network activity is idle.
That means that it blocks the execution of code, until the network activity is minimum. This guarantees that the actions performed by the ripper would not be lost due to unloaded content and that the state of a determined web page will not change during execution, until an action is performed by the ripper.
2.1.2 Exploration algorithm

**parameter:** Top-level-url

**Algorithm** DFS-GUIWEB()

```
access-top-level-url;

w ← window;

recursiveExploration(w, top-level-url, 0);

return;
```

**Procedure** recursiveExploration(w, url, depth)

```
if alreadyVisited(w) or notInDomain(url) or maxDepthLevelReached(depth) then
  return;
end

else

  l ← getListOfWidgets(w);
  E ← getExecutableWidgets(l);

  forall e in E do
    interactWith(e);
    D ← getNewWindows(e);

    foreach d in D do
      u ← getUrl(d);
      recursiveExploration(d, u, depth + 1);
    end
  end

Algorithm 1: DFS-GUI-WEB

The exploration algorithm was defined based on the work proposed by Memon, et al[1].
However some modifications were made to guarantee a correct execution in web applications. Firstly, the top level window is accessed through the top level url of the application under test. Secondly, the definition of widget was extended, as hyperlinks can also be considered widgets in web applications. In addition, a exploration criterion was added to guarantee that only the web pages that belong to the top level domain are explored. The exploration criteria also includes a validation to verify if the current window has already been visited. Finally, taking into account that really complex GUIs could cause a long time of execution a constraint to finish the exploration when a maximum depth level is reached was also added, in case that just some functionalities need to be tested. The pseudocode of the resulting algorithm can be observed in algorithm [1]

2.1.3 GUI state

As mentioned in the previous chapter, one of the most difficult challenges to develop a web application ripper is to define a GUI state. In order to be able to identify independent states, the first approximation was to identify each state by the structure of the DOM of the HTML. The DOM is an object model of HTML that defines each HTML element as an object with a set of properties. This means that each state is defined by the set of elements that compose the window, the order in which they are organized, and the properties of said elements.

Nevertheless, when evaluating this approximation, it was discovered that some transitions and events could trigger false positives when identifying new states, for example, when there are animations that change the size or the transparency of elements. To solve this problem, we decided to remove most HTML properties, leaving only the classes of the element and the id. As a consequence, in the final approximation, an state is defined by the set of elements that compose the window, the order of said elements and the set of properties (class, id) of each
element.

### 2.1.4 Oracle definition

A test Oracle is a component that determines the correctness of a program output \[1\]. In most of the projects that were mentioned in the prior work chapter, the oracle of the GUI ripping tools is crash based. However, detection of crashes on web applications is challenging as many actions can lead to crash. In addition, some broken links that could lead to states analogous to crashes can be easily misidentified as connection timeout errors.

Taking into account this, and the functionalities that Puppeteer offers, three types of errors were defined in the Oracle:

1. **Console errors**: Puppeteer can access the Chrome console and the JavaScript console of the browser. A console error is defined as any message that is thrown out by any one of this consoles with the error tag. Some examples are: *Failed to load resource, server responded with status of 400 (Bad request), Uncaught ReferenceError: JQuery is not defined, Uncaught TypeError: Cannot read property 'toString' of undefined* and JavaScript math errors.

2. **Console warnings**: A console warning is defined as any message that is thrown by either the Web console or the JavaScript console of the browser with the warning tag.

3. **Connectivity warning**: As mentioned above, it is not easy to identify broken links and network connection errors from one another. Therefore, this category should include any connectivity issue as a warning.
2.1.5 Model definition

The model that is extracted by the ripping tool, should be a multi model composed by the GUI model and the usage model of the web application. Taking this into account, the GUI ripping tool should be able to collect information of the exploration, while it simulates user interaction. The information collected should include the DOM structure, the url of each window, the errors defined in the Oracle section and the different interactions that are performed on each window. Additionally, screenshots of each window should also be collected. On the other side, in this project, user interactions will be simulated taking advantage of Puppeteer’s API, as it provides methods such as: `element.click()`, `element.tap()`, `keyboard.type()`, `element.focus()` among others.

Formally, the generated model is a graph $G=(V,E)$ where $V$ is a set of states in which each state has a unique structure of GUI elements, and $E$ is a set of edges, where each edge represents a state transition that is triggered by a user interaction.

2.1.6 Report design

Taking into account all the elements that have been defined, the report should include not only, a complete recollection of the errors that were found, but also a way for the tester to visualize the model in order to replicate the error. As a consequence of this, the report should have a interactive visualization, with all the relevant information that is captured during execution (url, screenshot, errors and warnings found). As well, the report should differentiate normal execution paths from error paths and the starting node from the others. Moreover, the report should also show the transitions that were executed with their respective data (type of interaction and screenshot of the widget).
2.2 Implementation

2.2.1 Selection of front-end frameworks

Before the development started, it was necessary to research about the different front-end frameworks and choose the ones that were going to be supported by the testing tool. Different types of web pages created with different frameworks, represent a great variety of factors that should be taken into account in order to create a GUI ripping tool. Taking this into account, the selection of the supported frameworks had to be made before we started writing the code. This with the purpose of delimiting the implementation from the beginning and fulfilling the coverage proposed in the objectives of this project.

Initially, the simplest way to develop a web application is using the static web page strategy. As the name suggests, a static web page is composed by static HTML documents that are stored and delivered to the users without much processing. Also, static web pages are usually simple and are common even though there are more recent and advanced web development strategies. Given that static web pages are still being used and that their simple logic don’t represent large complications in the implementation of the GUI ripping tool, we decided to support them in the development of RIPuppet.

Afterwards, we analyzed dynamic web pages and different frameworks that are used to create them. A dynamic web page construction is managed by an application server, so its contents are more variable and the rendering process is smarter. Nowadays, there are several front-end frameworks like React\footnote{https://github.com/facebook/react}, Angular\footnote{https://angular.io/} and Vue\footnote{https://vuejs.org/}. Each one of them has its own approach to the smarter rendering expected from dynamic web pages and this has an impact in the implemen-
tation of the GUI ripping tool. In the beginning, we thought that all these frameworks were very different in both development and rendering process. However, we realized that most of these modern frameworks use Server Side Rendering (SSR) to provide a better performance and experience to the end user. Given that Puppeteer has SSR methods to confront this issue, we could choose any of the frameworks without problems. Taking this into account, we decided to select React and Angular because we had more experience with the development components of these 2 frameworks and because they are ones of the most popular in the current market.

2.2.2 GUI exploration and interaction

Once the front-end frameworks were selected, the implementation process started using the elements designed in the previous state of the project.

Recursive exploration

The exploration algorithm is a key element of a GUI ripping tool, so the implementation of RIPuppet started with this component. The recursive exploration receives 2 parameters for its execution. The first one is the top level url that represents the page in which the exploration starts. Meanwhile, the second one is the levels of depth that the user wants to explore inside the web application. The latter is for having an additional ending condition and avoid an infinite recursive exploration. The execution starts with the testing tool using Puppeteer to visit the initial url and identifying the first state of the web application. When the first state is defined, the testing tool takes a screenshot of the current page, saves the DOM in a text file, stores any error found and marks the current state as already visited. Then it proceeds to get all the interactive widgets defined in the design phase using selectors with the HTML document. For example, the links were retrieved identifying the anchor elements \(<a>\) and the buttons were obtained
identifying the button elements `<button>` in the document. After this, the testing tool starts to interact with all these widgets and takes the resulting page from each interaction to call again the recursive algorithm. The process is repeated in each page, the testing tool determines if a new state was discovered and stores its corresponding information, including which interaction was used to get from the previous state to the current. This will continue until the ending criteria of the exploration is reached so it is important to mention that each time the algorithm is called, the counter that represents the current level of exploration is increased in order to stop the exploration when the max depth level is achieved.

**Identifying states**

During the exploration, it is important to check after each interaction whether a new state was discovered or not. This is delicate because not differentiating correctly the states could cause mistakes like visiting over and over the same page but marking each visit as a different state or missing new states due to their similarities to others already discovered. To avoid these problems, every time a new page is visited during the exploration, its DOM is directly compared with the DOMs stored from previous states and in case a match is found, the testing tool knows that the state was already visited. However, it is necessary to mention that the DOMs stored and compared during the exploration are not exactly the ones rendered in the web application. As we explained during the design phase, we decided to focus in the structure of each DOM to avoid false positives caused by attributes that are not relevant for identifying new states. For this reason, RIPuppet uses a HTML parser to extract from each DOM the elements required to identify each state and the result is stored as a representation for the different states. These files are the ones used for the comparisons and to determine the discovery of new states.
**Widget interaction**

The different widgets identified in the web pages require different types of interactions. Some are simpler like clicking links and buttons, but other ones are more complex like form filling with specific types of data. Puppeteer’s API offer different methods to interact with the page and it can simulate any interaction as if any user is navigating the web application. For the simpler interactions, there is no advanced logic because the transition between the states is achieved with a simple click using methods like `element.click()` or `element.tap()`. The real challenge is to simulate the more complex interactions like form filling, because it requires more logic to know how to fill each of the fields with coherent information.

**Form input automation**

In order to improve code coverage, a form automation component was added. This was done because, in order to discover new windows some forms must be completed with the correct data. This will be explained using an example: assume that there is a registration form in a web application. This registration requires that the user types his or her telephone. The text input field type is "tel", which is a type added in HTML5 that only receives numbers. Therefore, if the user or the ripping tool types a random string, the form would not be correctly completed and new windows would not be discovered.

To cope with this problem, this component identifies different HTML input components, extracts the type property of each one of the components, and generates the required input. To identify the input widgets and to extract their respective type properties, Puppeteer’s API is used. Once the type of each input element is determined, `faker.js`[^1] (a node library for random data generation) is called, to generate the required input. `faker.js` was selected as it allows offline

[^1]: https://github.com/marak/Faker.js/
data generation. This feature could be useful when testing locally deployed applications without network connection.

**Configuration file**

RIPuppet is configured through a file called `config.json`, it receives 5 parameters, and each parameter can modify the execution of the tool in different ways. An example of this file can be found in figure 2.2. This parameters are:

1. url: Defines the top level uniform resource locator (url) of the web application. This determines the starting state of the testing model.

2. headless: It is a boolean parameter. If the value is true, the testing tool will be executed in headless mode, otherwise, the execution will be headfull.

3. depthLevels: Defines the maximum depth level of exploration. When this level is reached, the DFS algorithm will return and would not continue exploring in depth.

4. inputValues: It is a boolean parameter, if the value is `false` the input fields will be filled
using faker.js. Otherwise, the personalized input feature will be activated and another parameter (values) will be needed. This feature will be explained next.

**Personalized input**

This feature was added to deal with the infeasible paths problem that was mentioned in the prior work section. The idea is that the user can personalize the values that are typed in the HTML input elements of the web page. This can be achieved by using the values parameter in the configuration file (see figure 2.2). The values parameter is a JSON object with a set of properties, where each property has a value. The first approach for this feature was that the key of each property was the HTML id of the input element and the value was what the tool should type in said input element. This way, each element could be identified uniquely and values typed could be highly customizable. However, when testing the tool against different web applications, we found that web developers usually do not include the HTML id property on all of the input fields, so not all input elements were filled. This led us to our final approach, in which the key of each JSON property in the values parameter is the placeholder of the input element that should be filled. This was decided, because we found that placeholders are more widely used than ids.

**Report visualization**

When the execution of RIPuppet finishes, a report will be generated to visualize the results of the exploration and the information of interest in each state. The report includes a graph that represents the exploration that the GUI ripping tool did through the web application. Each node represents the states identified and the transitions between the nodes represent the corresponding interactions to pass from one state to the other. Additionally, the graph uses a color scheme
to identify important nodes in the execution. The first node that represents the page where the exploration started has an orange color so it is easy to distinguish. Following the same logic, the nodes that represent states where errors or warnings were found are colored in red, so the tester can focus his attention in these nodes. Finally, the nodes that do not have errors or warnings and that are not the initial one, are colored in blue. This graph was developed using the d3.js\textsuperscript{5} library.

Furthermore, each node can be dragged around in order to move the graph and organize it in case the tester wants to see clearly an specific path. Also, each node can be clicked and the information of its corresponding state will be displayed. The report shows the screenshot of the state, the url and the amount of errors or warnings found in that particular state. Finally, there is more detailed information about the errors and warnings of each state. If a node with errors

\textsuperscript{5}https://d3js.org/
2.3 Evaluation

During the implementation process of this project, the GUI ripping tool was in constant evaluation in order to find possible mistakes. It was also compared against state of art tools like Cypress to identify similarities, differences and improvement opportunities.
2.3.1 Comparison with Cypress

Cypress\(^6\) is an end-to-end testing framework for web applications based on Javascript. It can be used to create tests for anything that runs in a browser and it is one of the most popular testing frameworks in the current market. When comparing Cypress and RIPuppet, it is possible to identify some similarities and some differences. On one hand, Cypress and RIPuppet both run properly on Chrome or Chromium, but they don’t support other major browsers like Safari, Firefox or Internet Explorer. Also, Cypress and RIPuppet are similar in the fact that they both offer the possibility to simulate GUI interactions and explore web applications. Finally, both Cypress and RIPuppet can take screenshots of the visited web pages, can be executed in headless or headful mode and are useful tools to test a web application during its development.

On the other hand, the biggest difference between Cypress and RIPuppet is that the former, as it was explained before, is a testing framework, meanwhile the latter is a testing tool already developed. Cypress can be used to develop different end-to-end tests, even GUI ripping tools similar to RIPuppet. But this implies expending time developing the tests as we did during the implementation of our GUI ripping tool. RIPuppet is a well-functioning ripper that already works with different web applications and it only needs to be executed in order to view the results in the generated report. In addition, given that RIPuppet was built in top of Puppeteer, its functionalities are limited by the capabilities of this Node library. Cypress offers additional functionalities that could be implemented in a GUI ripping tool that are not available in Puppeteer. For example, with Cypress is possible to record the execution of the tests and to take snapshots as the tests run in case there is a problem during the execution (Time travel).

\(^6\)https://www.cypress.io/
2.3.2 Identifying and correcting mistakes

Once the implementation process was finished, RIPuppet was tested with several open source web applications like Habitica\(^7\), WordPress\(^8\) and MantisBT\(^9\) with the objective of finding mistakes in our GUI ripping tool to correct them. After these tests, some issues were identified regarding connectivity and broken links. It was not possible to differentiate if the navigation to a link failed because the link was broken or because there was a problem with the internet connection. Our approach to solve this issue was to group these two situations in the same error category because we couldn’t assume that a link was broken if the internet connection failed.

In addition, we identified an issue related to pop ups in web applications. The problem was that when RIPuppet explored a web application that used excessively pop ups to show information, sometimes the pop ups could not be closed and it caused the exploration to be blocked. To solve this, we used different tools offered by Puppeteer’s API like the 'popup' event and the `dialog.dismiss()` method to identify any pop up or dialog during the exploration and close them when is needed. Finally, we did multiple corrections to the report implementation taking into account the recommendations from potential users of the testing tool. Among the improvements made to the reports, we stand out the inclusion of the color scheme to enhance the user experience.

\(^7\)https://habitica.com/
\(^8\)https://es.wordpress.com/
\(^9\)https://www.mantisbt.org/
Chapter III

Conclusions

After finishing the development and evaluation of the testing tool, the goals that were set for this project were fulfilled. Still, some challenges remain and the work could be extended as it provides numerous opportunities.

3.1 Limitations

- The testing tool can only be executed in Chrome or Chromium, as it is built on top of Puppeteer. This presents a limitation when it is necessary to test a web application on different browsers.

- Puppeteer is still a "young" tool, because of this it is constantly updated (During the development of this project three Puppeteer releases were made), this can cause instability. As a consequence, the tool requires constant maintenance.

- As mentioned in the development sections, there was not an approach that allowed the tool to differentiate broken links, from connectivity errors so this errors were grouped in
one category. This presents a limitation if one of the purposes of the execution is to find broken links.

### 3.2 Future work

- As the tool is build on top of Puppeteer, it could be extended to profile web applications during execution using Chrome devtools protocol.

- The tool could also be extended to simulate mobile browsers, as Puppeteer provides mobile-like (tap, swipe) interactions and provides functions to modify the browser viewport size.

- The tool could be extended to test secure connections. This can be done by using puppeteer `SecurityDetails` class.

- The tool can be extended to support continuous integration engines, as this is a feature that is supported by state of the art tools like Cypress.

- The definition of state can still be refined, as some elements like ads can modify the structure of the page, generating false positives that are detected as new states.
References


