Morpheus Web Testing: A Tool for Generating Test Cases for Widget Based Web Applications

Romulo de Almeida Neves*, Willian Massami Watanabe and Rafael Oliveira

Federal Technological University of Parana (UTFPR), Cornélio Procópio, Paraná, Brazil
E-mail: romulo.neves@gmail.com
**Corresponding Author*

Received 07 October 2020; Accepted 25 August 2021;
Publication 22 December 2021

Abstract

Context: Widgets are reusable User Interfaces (UIs) components frequently delivered in Web applications.In the web application, widgets implement different interaction scenarios, such as buttons, menus, and text input.

Problem: Tests are performed manually, so the cost associated with preparing and executing test cases is high.

Objective: Automate the process of generating functional test cases for web applications, using intermediate artifacts of the web development process that structure widgets in the web application. The goal of this process is to ensure the quality of the software, reduce overall software lifecycle time and the costs associated with tests.

Method:We elaborated a test generation strategy and implemented this strategy in a tool, Morpheus Web Testing. Morpheus Web Testing extracts widget information from Java Server Faces artifacts to generate test cases for JSF

Journal of Web Engineering, Vol. 21_2, 119–144.
doi: 10.13052/jwe1540-9589.2121

web applications. We conducted a case study for comparing Morpheus Web Testing with a state of the art tool (CrawlJax).

Results: The results indicate evidence that the approach Morpheus Web Testing managed to reach greater code coverage compared to a CrawlJax.

Conclusion: The achieved coverage values represent evidence that the results obtained from the proposed approach contribute to the process of automated test software engineering in the industry.

Keywords: User interfaces, widgets, morpheus web testing, code coverage.

1 Introduction

The User Interface (UI) is an essential part of most applications in use. Currently, there are three UI types: Graphical User Interface (GUI), used in desktop application; Web User Interface (WUI), used in web applications; and Handheld User Interface (HUI), used in mobile devices [21]. The UIs are frequently composed by elements (widgets) that can be reused in different [19, 21], such as frames, buttons, menu items, and text box [15]. On the other hand, software testing is one of the main activities performed to improve the quality of the software under development. Its main objective is to detect errors as early as possible in the software development cycle [16], in order to minimize the cost of delivering fixes to the product [18, 20]. Delamaro et al. (2007) state that a successful testing approach can decrease testing effort, contribute to improving product quality and reduce the costs of maintenance [7].

In some development scenarios, tests are performed manually and the cost associated with elaborating test cases is high, given the high number of possible user interaction scenarios with applications [17]. In these scenarios, the test activity can become costly and misleading [1]. Other aggravating factors for performing the tests are the high complexity of the systems currently developed and their continuous evolution [2, 12]. Thus, software testing automation is essential to improve the efficiency of software testing activity and has become one of the alternatives to obtain a product with a reduced number of defects [8,11] and to reduce costs, increase test efficiency, ensure software quality and reduce overall software lifecycle time [9].

This paper has the goal of elaborating strategies for generating automated functional test cases for web applications. We elaborated a test case

generation strategy that extracts widgets information from UI configuration artifacts and uses this information to generate functional tests. We developed a tool, Morpheus Web Testing, which implemented this strategy using two WUI component frameworks: the JavaServer Faces (JSF) and Primefaces. Our strategy focuses on the generation of system-level functional test cases. A case study was conducted to evaluate how our test case generation strategy performs in comparison to another state-of-art technique. Moreover, the case study was conducted in an industrial setting, with a production ready JSF/Primefaces based web application.

The remainder of this article is organized as follows: Section 2 presents related works. Section 3 presents the proposal of this paper. Section 4 presents the evaluation of our proposed test generation strategy. Section 5 presents discussions. Finally, Section 6 presents final considerations, limitations of this work and future works.

2 Related Work

In [15], the authors presented a UI's testing framework denominated GUITAR which generates test cases based on events implemented in the UI. The framework generates application models from Java UI artifacts which are, then, used in the testing of the application. In the test cases generation process, GUITAR extracts information about the structure of all windows, widgets, as well as their attributes and graphical interface events. The framework creates a flow of events with all possible UI event interactions. These events are used to generate UI test cases that are sequences of UI events. GUITAR also supports the execution of the generated test cases in a Java UI application.

In [14] the authors presented a tool called AutoBlackTest. This tool implements a process that incrementally generates test cases as a user interacts with the application. This generation process is divided in two steps. In the first step, the tool generates a sequence model of events that can be produced through an interaction with the application UI under test. It is worth mentioning that model generation occurs through the use of a reinforcement learning system called Q-learning. Lastly, in the second step, it begins the generation of a data set of tests that covers the sequences in the model.

In [23], the authors presented a tool called CrawlJax. This tool performs the test case generation process automatically, analyzing state changes in the web application interface with Ajax (Asynchronous JavaScript and XML). This process is divided into two steps: (i) a crawler (controller) that exercise

client-side code and identifies the clickable elements that change state within the DOM (DOM stands for Document Object Model) built dynamically in the browser. Finally in the second step (ii) the creation of a graph of state flow, called the SFG (State-Flow Graph) that captures dynamically DOM states, the UI states, and the possible transitions made between them.

In [6], the development of the WebMate tool is reported. WebMate is a tool that performs test case generation for web applications. WebMate explores the functionality of a web application that detects the differences between web browsers and operating systems. The process test case generation in the WebMate consists of three steps: (i) the information is extracted an URL wherein the user interacts with the application by examining all buttons, links, forms, or any element manipulated by events which can trigger off interaction with the user. In (ii), usage models are extracted from the application in a graph form, where the nodes correspond to the different application states and the edges represent user interactions. In (iii), test are performed in the Web application, exercising all transitions of the generated usage model, verifying the compatibility between browsers, conducting code analysis and regression tests.

In [24], the authors proposed an approach to obtain textual input values while testing Android apps automatically. This process consists of four steps:(i) Description matching, in this step is identified and matched descriptive labels with input fields concept extraction, input value acquisition and input value consumption. (ii) Concept extraction, in this step, natural language processing techniques are used for extracting the concept associated with the label. (iii) Input value acquisition, in this step the concepts are used to query a knowledge base for candidate input values. (iiii) Input value consumption clustered, in this step, the UI elements are filled according to their functionality into input and actions, filling the input elements first and then interacting with the actions.

In [10], the authors performed a survey that assists software developers in making a decision regarding the testing tools, based on black and white box approaches to use in web/mobile applications. Thereby a set of current testing approaches were surveyed using four test-key factors, (i) Artificial Intelligence (AI), (ii)Security focused, (iii) Fully automated and (iv) Heuristic search.

Differently from these prior works [5, 15, 23] that used low-level interface components based on HyperText Markup Language (HTML) to generate test cases, our work extracts information from complex interface components, widgets, as defined in the web application. Our test case generation strategy

was implemented in a tool and uses Extensible HyperText Markup Language (XHTML) defined in a UI framework for web development: JSF and Primefaces. In this context, this paper reports on a investigation of whether this approach of using higher abstraction level components (widgets defined in JSF/Primefaces artifacts) for generating functional test cases can enhance the test case generation process. We identified that the use of the higher level interface components in the JSF/Primefaces components presents the following advantages: (i) the possibility to get more information than an HTML file and thus predict and improve the interaction levels of the components, and (ii) the possibility of generating more test case inputs and, consequently, achieve greater code coverage.

3 Test Case Generation Approach

This paper had the goal of generating automatic functional test cases for web applications. In order to achieve this goal, we elaborated a test case generation strategy, which extracts widget semantic information from software artifacts that define the UI of web applications. As a proof-of-concept, we implemented this test case generation strategy in a tool called Morpheus Web Testing which consists of test case generation using two frameworks that assist developers in web application building: JSF and Primefaces.

In Morpheus Web Testing, the test case generation strategy was divided into four steps, as shown in Figure 1.

In the first step, Entry (see Figure 1(A)), Morpheus Web Testing receives as an input a project that uses the JSF and Primefaces frameworks. After the project entry, the process of generating of usage model is started, extracting and analysing the Widgets defined in the XHTML code of JSF projects as a UI definition model (see Figure 1(B)). In this second step, web application information about the structure of all windows, widgets as well as their attributes and events of interface graph are extracted. These events are,

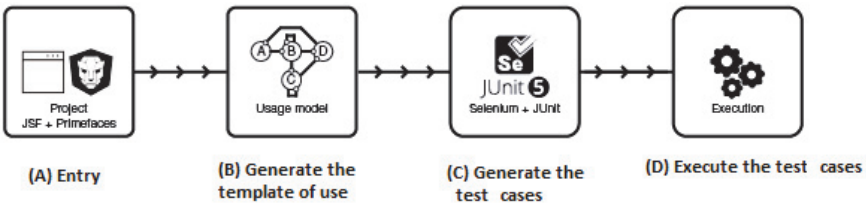


Figure 1 Proposal of the test model and development of this paper.

then, used to generate test cases from the WUI that are sequences of WUI events. Finished the second step, the generation of test cases is started (see Figure 1(C)), in which JUnit and Selenium WebDriver based test cases are generated, as output. Finally, in the last step (see Figure 1(D)), the test cases can be executed.

Selenium is an open source testing tool which isused to automate the test cases and enhance the testing performance. Selenium is an automated testing tool for web application. Selenium WebDriver basically work in two ways first locate the element and then perform some action on them. Selenium WebDriver locate element by usingid, name,Xpath, CSS, link text, partial link text. Selenium provides a rich set of functions which is used to testing of a web application [13].

Test cases can be defined as a way of establishing the inputs to be informed by the tester (manually or with tool support) and the results expected from this action. The test cases are composed of three steps: input, steps and oracles [7, 22].

In order to generate the input of a test case, Morpheus Web Testing identifies the objects and the properties of a WUI. Then, Morpheus Web Testing uses different strategies for each type of input (widget), as demonstrated in Table 1 and detailed below.

- For widgets of type InputTextArea, InputNumber, InputMask, InputText, Password, TextEditor, CKEditor, or Editor, multiple text input scenarios are generated, such as: randomly generated numbers/texts and blank input;
- For widgets of type: Button, CommandButton, CommandLink, Link and LinkButton, test cases in which clicks are performed on these components are generated;
- For widgets of type SelectBooleanButton, SelectBooleanCheckBox, SelectOneButton, SelectOneRadio, SelectOneMenu, SelectOneListBox, SelectManyButton, SelectManyMenu or SelectManyCheckBox, multiple test cases which select each possible input are generated; and
- For widgets of type calendar, test cases which insert random dates, integers, and empty strings are generated.

For the strategies of insertion of texts and random integers, if the maxLenght property is not specified in the test cases will consider a default a size of 50 characters for the component. The test case generation process is performed through instructions defined in templates. In [4] templates are defined as text files, instrumented with selection and expansion conditions of

Table 1 Components and strategies used by Morpheus Web Testing

Component	Strategy					
	IRIN	IRGT	TIB	Click	Selection	IRD
InputTextArea	X	X	X			
InputNumber	X	X	X			
InputMask	X	X	X			
InputText	X	X	X			
Passoword	X	X	X			
TextEditor	X	X	X			
Editor	X	X	X			
ckEditor	X	X	X			
Button				X		
CommandButton				X		
ComandLink				X		
Link				X		
LinkButton				X		
SelectBooleanButton					X	
SelectBooleanCheckBox					X	
SelectOneButton					X	
SelectOneMenu					X	
SelectOneListBox					X	
SelectManyButton					X	
SelectManyMenu					X	
SelectManyCheckBox					X	
calendar	X		X			X

IRIN = Insertion of Random Integer Numbers
IRGT = Insertion of Randomly Generated Text
TIB = Text Insertion in Blank
IRD = Insertion of Random Dates

code. These instructions are responsible for consulting an input that can be a program, a textual specification or diagrams and as a result, it's possible to get the parameter to produce the source code [3].

Figure 2 illustrates an example of the test case generation process. This figure illustrates a test case generation for the inputText widget in line 2 of Figure 2(A), in the XHTML file. The widget definition markup inside the XHTML file is used to identify the attributes of the widget (id, max length and type of the widget) and generate test case scenarios associated to that type of widget. The example illustrates the generation of a test case which inserts a random string as text input for the widget.

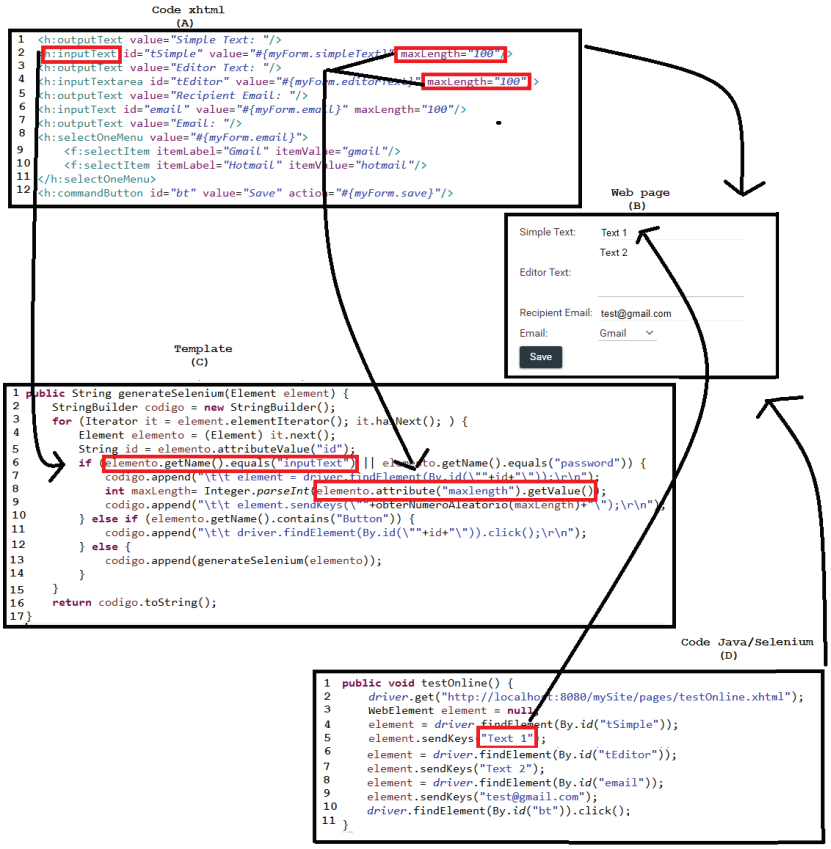


Figure 2 Java code generation with Selenium framework based on template.

Figure 3 illustrates a snippet of Java code generated by Morpheus Web Testing (see Figure 2) wherein, for each widget in the XHTML file, a specific input will be generated. In JSF XHTML templates, input type widgets must be placed inside a form element, hence, after generating the input test cases for the input widgets, Morpheus Web Testing generates an action for activating their respective form element, clicking the submit button of the form.

The Morpheus Web Testing approach is an open source software is available for access through the address.[1] Furthermore was developed using the following frameworks:

[1] https://github.com/raneves/morpheus.git

```
1  public void instrutor() {
2      driver.get("http://localhost:8080/helpDesk/instructor.xhtml");
3      WebElement element = null;
4      element = driver.findElement(By.id("name"));
5      element.sendKeys("a text");
6      element = driver.findElement(By.id("email"));
7      element.sendKeys("e@gmail.com");
8      element = driver.findElement(By.id("phone"));
9      element.sendKeys("9999999999");
10     element = driver.findElement(By.id("cell phone "));
11     element.sendKeys("888888888");
12     element = driver.findElement(By.id("login"));
13     element.sendKeys("myLogin");
14     element = driver.findElement(By.id("password"));
15     element.sendKeys("nquwehuero");
16     driver.findElement(By.id("btSave")).click();
17     driver.findElement(By.id("btClean")).click();
18 }
```

Figure 3 Exemple of Java code generated.

- **Java**: JDK version 1.8; and
- **Dom4j**: An open source Java library for parsing the XHTML code, identifying widgets in the web application.

4 Evaluation

In order to evaluate our test case generation approach, we conducted a case study for evaluating the quality of the test cases which were generated. In our study, the quality of test cases was measured in terms of the Code Coverage metric. The case study design was guided by the following hypotheses $\mathbf{H}_0$ and $\mathbf{H}_1$.

- $\mathbf{H}_0$: Code coverage obtained by test cases generated using widget information extracted from WUI definition (our approach) does not differ from state-of-art test cases generation approaches.
- $\mathbf{H}_1$: Code coverage achieved by test cases generated using widget information extracted from WUI definition (our approach) is superior to state-of-art test cases generation approaches.

4.1 Methodology

Considering the hypotheses $\mathbf{H}_0$ and $\mathbf{H}_1$ previously described, the methodology of this study was designed to compare how our test case generation

approach performs in comparison with other state-of-art test case generation tool, in regards to the Code Coverage metric. Our approach was implemented in Morpheus Web Testing tool, and it focuses on web applications. The baseline of our case study (control) was the CrawlJax tool, which represents a state-of-art open source tool for generating test cases for web applications. However, differently from Morpheu Web Testing, CrawlJax uses semantic information extracted from the HTML source-code of a web application. Hence, CrawlJax does not rely on widget information, since these information are not defined in the HTML code.

For this study, we used a web project called Exactus CRM (a commercial application with closed source code) and, two automatic test case generation tools for web applications, Morpheus Web Testing, and CrawlJax (state-of-the-art technique), with the objective of performing a comparative evaluation between both applications. Morpheus Web Testing is a tool which uses widgets information extracted from JSF XHTML code to generate test cases. Exactus CRM is an industrial, production-ready JSF/Primefaces web application, which implements client support, finances and accountability management functionalities. It was developed by Exactus,[2] a brazilian software development company, has 11,168 active users, was first deployed in production in 2012 and is currently deployed in Jelastic[3] Platform-as-a-Service infrastructure. Exactus CRM was built using the following technologies:

- **JSF**: version 2.2.8;
- ***Primefaces***: version 6.0;
- ***Hibernate - Java Persistence API* (JPA)**: version 5.0;
- ***Maven***: version 3.1;
- **Cobertura**: version 2.1;
- ***Jetty***: version 9.4.9;
- **Java**: JDK version 1.8;
- ***selenium-java***: version 3.7.1; and
- **MySql**: version 5.5.

Morpheus Web Testing was first designed to reduce costs associated to test case elaboration and execution for Exactus CRM. Exactus CRM architecture was designed accordingly to the Model-View-Controller Design Pattern and is divided into 4 layers described next:

[2]https://www.exactus.com.br/
[3]https://jelastic.com/

- **Data access object layer (DAO)**: contains all access and execution logic for the database;
- **Plain old java object layer (POJO)**: maps classes to tables in the database, that is, a JPA the entity represented by the design pattern named JavaBeans (the class should have private attributes, a default constructor without arguments and methods getters and setters for each attribute);
- **ManagedBean layer**: contains codes responsible for performing the back-end, business rules and validation in general. This layer receives the entered data through the XHTML pages, processes and returns the results of the operation to the page; and
- **Utilities layer**: contains code that validates different parameters such as: Brazilian numeric IDs, passport code, emails, zip code, and phone.

Exactus CRM contains a total of 207 classes and 11415 lines of code. The number of classes and the total lines of code for each layer are described next:

- **DAO layer**: 65 classes and 1618 line of code;
- **ManagedBean layer**: 39 classes and 4857 line of code;
- **Utilities layer**: 4 classes and 171 line of code; and
- **POJO layer**: 99 classes and 4769 line of code.

After configuring Morpheus Web Testing and CrawlJax for generating test cases for Exactus CRM, both approaches were executed. Their execution were run in an instrumented version of Exactus CRM, which collected Coverage Metrics for all classes of the web application. Next section, presents the results of this methodology.

4.2 Results

In this study, Morpheus Web Testing generated 91 test cases. Comparing the results between our approach and the state-of-the-art can provide evidences supporting hypothesis H_0 or H_1. Next, we compare the Code Coverage by line and branch results for each approach separately, according to each layer of Exactus CRM, then we provide a general comparison considering all layers:

4.2.1 DAO layer

In the DAO layer, 1062 lines (out of 1618 existing) were covered by test cases generated by Morpheus Web Testing, totaling 66% coverage. While for the CrawlJax approach, 543 lines (out of 1618 existing) were covered,

totaling 34% coverage. Grouping the coverage results for each class of the DAO layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that coverage with both approaches do not match a normal distribution, with w = 0.87864 and p-value = $1.196e^{-05}$ for Morpheus Web Testing, and w = 0.89158 e p-value = $3.438e^{-05}$ for CrawlJax. Then, Mann Whitney Wilcoxon statistical test showed significant differences between the coverage of both approaches with W=26042 and p-value = 0.0001478; The probability density of the coverage reported for the DAO layer using Morpheus Web Testing and Crawljax is illustrated in Figure 4(A).

In regards to branch Code Coverage, 90 branchs (out of 158 existing) were covered by test cases generated by Morpheus Web Testing, totaling 56% branch. While for the CrawlJax approach, 44 branchs (out of 158 existing) were branched, totaling 27% branch. Grouping the branch results for each class of the DAO layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that branch with both approaches match a normal distribution, with w = 0.96031 and p-value = 0.78826 for Morpheus Web Testing, and w = 0.86875 e p-value = 0.06301 for CrawlJax. Then, Mann Whitney Wilcoxon statistical test showed significant differences between the coverage of both approaches with V = 35 and, p-value = 0.02055. The probability density of the branch reported for the DAO layer using Morpheus Web Testing and Crawljax is illustrated in Figure 5(A).

4.2.2 POJO layer

In the POJO Layer, 1200 lines (out of 4769 existing) were covered by test cases generated by Morpheus Web Testing, totaling 25% coverage. While for the CrawlJax approach, 1081 lines (out of 4769 existing) were covered, totaling 23% coverage. Grouping the coverage results for each class of the POJO layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that coverage with both approaches do not match a normal distribution, with w = 0.848242 and p-value = $7.137e^{-05}$ for Morpheus Web Testing, and w = 0.92879 e p-value = $4.637e^{-05}$ for CrawlJax. Then, Mann Whitney Wilcoxon statistical test did not show significant differences between the coverage of both approaches with W = 5122.5 and p-value = 0.5825; The probability density of the coverage reported for the POJO layer using Morpheus Web Testing and Crawljax is illustrated in Figure 4(B).

In the regards to branch Code Coverage, 74 brachs (out of 390 existing) were branch by test cases generated by Morpheus Web Testing, totaling 185% branch. While for the CrawlJax approach, 72 lines (out of 390 existing) were branched, totaling 18% branch. Grouping the branch results for each class of the POJO layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that branch with both approaches do not match a normal distribution, with w = 0.70793 and p-value = $5.008e^{-06}$ for Morpheus Web Testing, and w = 0.71375 e p-value = $6.059e^{-06}$ for CrawlJax.Then, Mann Whitney Wilcoxon statistical test did not show significant differences between the coverage of both approaches with V = 1, p-value = 1. The probability density of the branch reported for the POJO layer using Morpheus Web Testing and Crawljax is illustrated in Figure 5(B).

4.2.3 ManagedBean layer

In the ManagedBean Layer, 2295 rows (out of 4857 existing) were covered by the test cases generated by Morpheus Web Testing, totaling 48% of coverage. While for the CrawlJax approach, 2234 lines (out of 4857 existing) were covered, totaling 46% coverage. Grouping the coverage results for each class of the ManagedBean layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that coverage with both approaches do not match a normal distribution, with w = 0.92668 and p-value = 0.01408 for Morpheus Web Testing, and w = 0.89555 e p-value = 0.001648 for CrawlJax. Then, Mann Whitney Wilcoxon statistical test did not show significant differences between the coverage of both approaches with W = 854.5 and p-value = 0.3495; The probability density of the coverage reported for the ManagedBean layer using Morpheus Web Testing and Crawljax is illustrated in Figure 4(C).

In the regards to branch Code Coverage, 531 branchs (out of 121 existing) were branched by the test cases generated by Morpheus Web Testing, totaling 43% of branch. While for the CrawlJax approach, 504 branchs (out of 1212 existing) were branched, totaling 41% branch. Grouping the branch results for each class of the ManagedBean layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that branch with both approaches match a normal distribution, with w = 0.95113 and p-value = 0.1228 for Morpheus Web Testing, and w = 0.94319 e p-value = 0.07004 for CrawlJax. Then, no significant differences were observed according to a Student T-test between the coverage of both approaches t = 15.451, df = 69, p-value $< 2.2e^{-16}$. The probability

density of the branch reported for the ManagedBean layer using Morpheus Web Testing and Crawljax is illustrated in Figure 5(C).

4.2.4 Util layer

In the Util Layer, 109 lines (out of 171 existing) were covered by test cases generated by Morpheus Web Testing, totaling 64% coverage. While for the CrawlJax approach, 77 lines (out of 171 existing) were covered, totaling 46% of coverage. Grouping the coverage results for each class of the Util layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests did not show evidence that coverage with both approaches match a normal distribution, with w = 0.95568 and p-value = 0.7518 for Morpheus Web Testing, and w = 0.84161 e p-value = 0.2001 for CrawlJax. Then, no significant differences were observed according to a Student T-test between the coverage of both approaches with t = 2.2479, df = 4.272 and p-value = 0.08358; The probability density of the coverage reported for the Util layer using Morpheus Web Testing and Crawljax is illustrated in Figure 4(D).

In the regards to branch Code Coverage, 11 branchs (out of 38 existing) were branched by test cases generated by Morpheus Web Testing, totaling 39% branch. While for the CrawlJax approach, 14 branchs (out of 38 existing) were branched, totaling 36% of branch. Grouping the branch results for each class of the Util layer, for each approach we run a Shapiro-Wilk normality test. The results of these tests did not show evidence that coverage with both approaches match a normal distribution, with w = 0.90977 and p-value = 0.4173 for Morpheus Web Testing, and w = 0.98047 e p-value = 0.7322 for CrawlJax. Then, no significant differences were observed according to a Student T-test between the coverage of both approaches with t = 4.3201, df = 5, p-value = 0.00757. The probability density of the branch reported for the Util layer using Morpheus Web Testing and Crawljax is illustrated in Figure 5(D).

4.2.5 All layers

In the All Layers, 4641 rows (out of 11415 existing) were covered by the test cases generated by Morpheus Web Testing, totaling 41% coverage. While for the CrawlJax approach, 3935 lines (out of 11415 existing) were covered, totaling 35% coverage. Grouping the coverage results for each class of all layers of Exactus CRM, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that coverage with both approaches do not match a normal distribution, with w = 0.93908

and p-value = $1.262e^{-07}$ for Morpheus Web Testing, and w = 0.90323 e p-value = $2.445e^{-10}$ for CrawlJax. Then, significant differences were observed according to a Mann Whitney Wilcoxon test between the coverage of both approaches with w = 26042 and p-value = 0.0001478; The probability density of the coverage reported for the all layers using Morpheus Web Testing and Crawljax is illustrated in Figure 4(E).

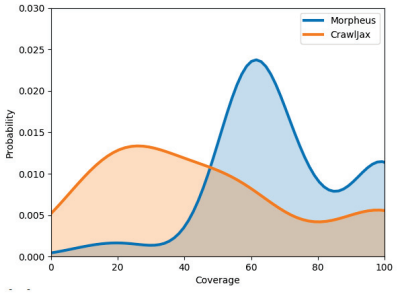
In the regards to branch Code Coverage, 710 branchs (out of 1798 existing) were branched by the test cases generated by Morpheus Web Testing, totaling 39% branch. While for the CrawlJax approach, 634 branchs (out of 1798 existing) were branched, totaling 35% branch. Grouping the branch results for each class of all layers of Exactus CRM, for each approach we run a Shapiro-Wilk normality test. The results of these tests showed evidence that coverage with both approaches do not match a normal distribution, with w = 0.87319 and p-value = $1.562e^{-06}$ for Morpheus Web Testing, and w = 0.89451 e p-value = $1.0225e^{-05}$ for CrawlJax. Then, significant differences were observed according to a Mann Whitney Wilcoxon test between the coverage of both approaches with V = 206 and p-value = 0.0001705. The probability density of the branch reported for the all layers using Morpheus Web Testing and Crawljax is illustrated in Figure 5(E).

When the sample has a tendency to follow a normal distribution, a parametric test is applied Student T-test and in the opposite case, ie non-normality, wilcox is applied. Shapiro-Wilk normality tests were used for identifying if a sample is normal or not normal, if its p value is less than 0.05 it means that there are indications that the sample is not normal.

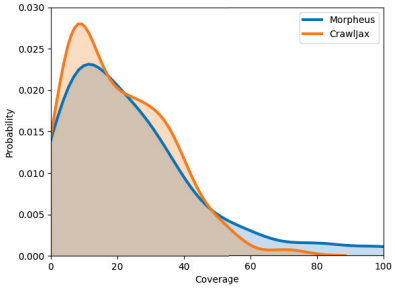
In Figure 4, it is possible to observe that the coverage distribution was greater in Morpheus, mainly in the DAO, UTIL and all layers. At the same time it is also possible to observe that for certain samples (layer util) the coverage distributions of the approaches are similar to a normal distribution, on the other hand samples in the layers (Pojo layer, dao and all layers) do not ressemble a normal distribution.

In Figure 5, it is possible to observe that the coverage distribution was greater in Morpheus, mainly in the DAO, UTIL and all layers. At the same time it is also possible to observe that for certain samples (DAO layer and ManagedBean) the coverage distributions of the approaches are similar to a normal distribution, on the other hand samples in the layers (Pojo layer, util and all layers) do not ressemble a normal distribution.

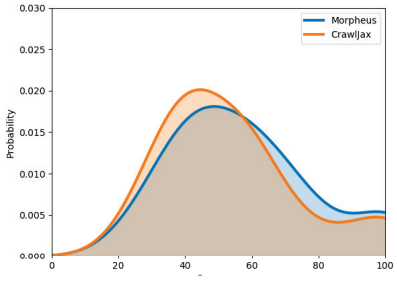
All statistical tests were conducted considering a 0.95 confidence interval. In regards to the coverage results of both approaches, the test cases generated by Morpheus Web Testing and CrawlJax failed to reach ManagedBean and



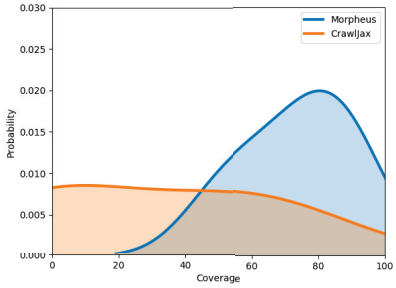
(A) Coverage probability density for the DAO layer using Morpheus Web Testing and Crawljax



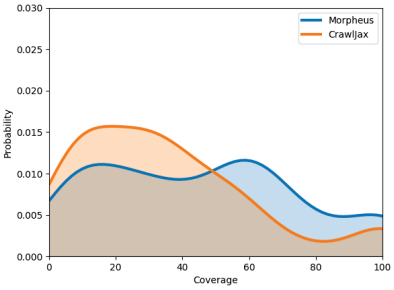
(B) Coverage probability density for the POJO layer using Morpheus Web Testing and Crawljax



(C) Coverage probability density for the ManagedBean layer using Morpheus Web Testing and Crawljax



(D) Coverage probability density for the Util layer using Morpheus Web Testing and Crawljax



(E) Coverage probability density for the all layers using Morpheus Web Testing and Crawljax

Figure 4 Coverage probability density using Morpheus Web Testing and Crawljax.

POJO get and set methods which were not explicitly included in the WUI of the web application. Moreover, exception handling routines implemented in the Util and DAO layers were not reached by the approaches as well. Future works should include ways for testing these components.

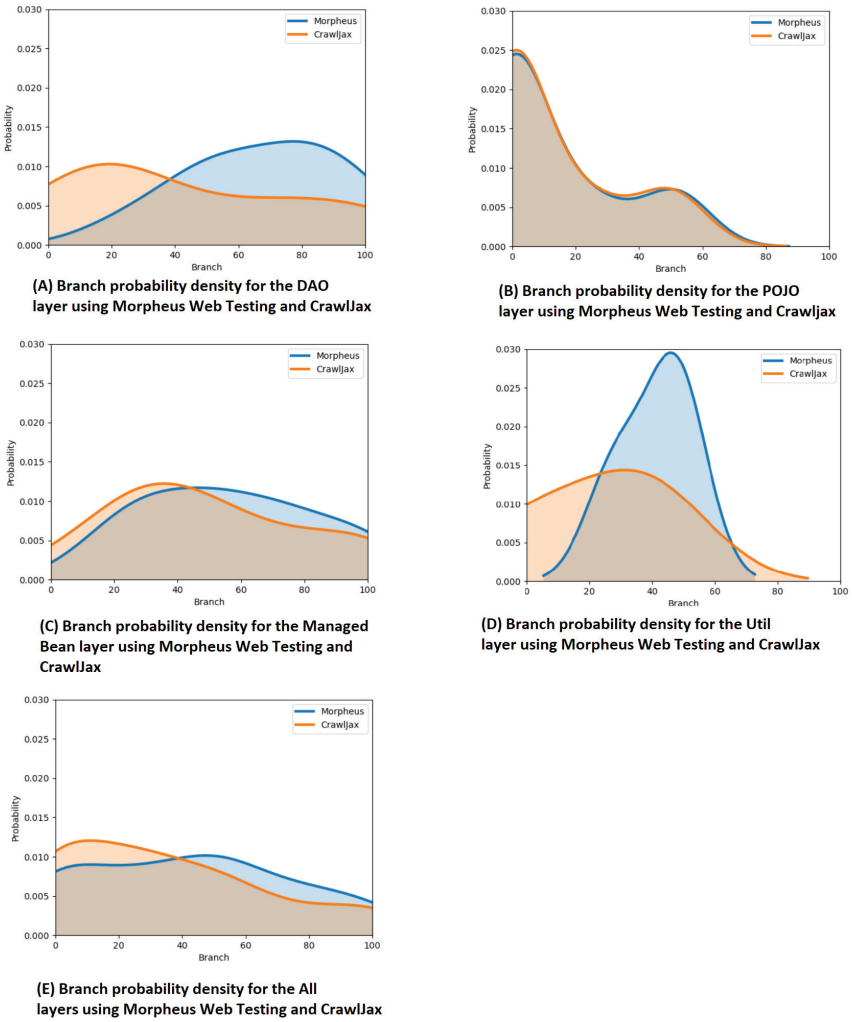


Figure 5 Branch probability density using Morpheus Web Testing and Crawljax.

Considering the case study scenario, the results represent evidences that support hypothesis H_1, considering that all coverage results were higher when using Morpheus Web Testing in comparison to Crawljax. Furthermore, H_1 was also supported by the significant differences observed when comparing the coverage of both approaches. More specifically, it was observed that Morpheus Web Testing coverage results impacted significantly the coverage of the DAO layer of the web application.

5 Discussion

This paper presented a strategy for generating functional test cases for web applications. We proposed a generation strategy which uses widget information for generating test cases. The test case generation strategy was implemented in Morpheus Web Testing tool which used widget information available in XHTML JSF/Primefaces templates for generating the test cases. Our general hypothesis was that using widget information, which stand for complex UI components, could enhance the coverage of the test case generation process for web applications, in comparison to state-of-art approaches which used HTML markup for generating test cases. We conducted a case study for comparing the coverage obtained when executing the test cases generated by our approach and test cases generated by a state-of-art approach, CrawlJax.

The CrawlJax execution obtained an average of 35% coverage, while the Morpheus Web Testing generated test cases averaged 41% coverage. The results observed in the case study showed statistical differences between both approaches, and thereby, bringing evidence to support the alternative hypothesis $\mathbf{H}_1$. One of the factors that contributed to the approach of the Morpheus Web Testing achieved a coverage greater compared to CrawlJax, occurred due to the use of the interface components XHTML in the process of generating test cases that have been defined by the JSF framework together with the components of Primefaces interface.

Separately analysing the layers of the web application used in the case study, the DAO layer showed higher significance in the coverage comparison between both approaches. The factors that determined this disparity was that when identifying the page components XHTML, the Morpheus Web Testing approach synthesizes multiple test case input scenarios, while the CrawlJax approach triggers clicks about the links rather than navigating within complex widgets or inserting different values for form input elements.

CrawlJax and WebMate use HTML markup for generating events that are associated with each component. However, Morpheus Web Testing does not use the markup generic of HTML, but as it is using the JSF XHTML templates, there is more information than a HTML file. Given that in the of Morpheus Web Testing analyses the WUI in a higher abstraction level, through the XHTML template, we argue that it can identify menu or a calendar widget, while other approaches can only identify a DIV or an INPUT HTML element.Having identified a menu, Web Testing can predict a type of interaction that a div would not predict, such as: with a menu, it would be

necessary a simple click, with right or left button to open other actions.Thus, given we used complex widgets in the web application used in the case study, Morpheus Web Testing was capable of identifying more states of the application, achieving a higher coverage than CrawlJax.

AutoBlackTest [14], uses two strategies to carry out the process of generating test cases (i) the use of a learning technique named Q-learnig, and (ii) multiple heuristics to address some common cases such as: filling out forms and file persistence. On the other hand, Morpheus Web Testing uses the semantics of XHTML interface components in the test case generation process. It should be noted that, to effectively compare our work with AutoBlackTest [14], further experiments are required and could be realized in future work.

Crawljax and Morpheus tools still have room for improvement and that, in future works we could link the generation of test cases using the types of inputs generated by other approaches as (The Link [24], AutoBlackTest [14] and WebMate [6]) to in order to improve coverage, potentially improving coverage in DAO and ManagedBean Layers.

In the area of test case generation for web applications, two related studies (CrawlJax [23] and WebMate [6]) report the use of the structure HTML of a web page to generate the test cases. HTML pages have a limited set of interface components by default: text input, buttons, links, among others. Morpheus Web Testing, on the other hand, was implemented to use complex interface components (widgets) from JSF/Primefaces XHTML templates to generate test cases. In these templates, interface components are specified using higher level of abstraction definitions such as for example calendar widget, inputs with validation fields, links and associated buttons to input, among others.

In this work, we investigated whether the use of interface components at a higher abstraction level such as menus could generate greater coverage in test case generation strategies compared to the use of HTML interaction elements.Our general research hypothesis was that, when a calendar widget is identified, associated with a text field, specified in the XHTML template with the Primefaces framework, it is possible to predict more elaborate levels of interaction (considering the relationship between components, validation strategies, and message fields) compared to separate identification of HTML elements, without relationship between widget links and inputs and no identification of validation strategies in XHTML with the Primefaces framework. Hence, we argue that using higher abstraction level WUI definition can enhance the process of test case generation considering that: (i) get more

information than an HTML file and thus predict and improve component interaction levels, and (ii) the ability to generate more functional test cases and thereby achieve greater code coverage.

In the Morpheus Web Testing approach, not all possibilities of interface components have been exercised. The components that were considered in the process of the test case generation process were: input with masks and validation messages; calendar widget; checkbox, links, combobox, text area, radio buttons. For each widget type, it has been established a set of test cases many different possibilities of input, considering the mechanisms of the specific interaction of each widget and their integration in the web application. Morpheus Web Testing focused the test case generation process for these widget, given they were the types of widget used in Exactus CRM.

The technique used was the generation of input for test cases, within this, the oracles were not generated, so the faults are not revealed, that is, only the application is explored. In this case if we consider detecting the number of failures identified by approaches, others study should be carried out.

Separately analysing the web application layers, can provide different insights per layer. Crawljax and Morpheus use different software artifacts for generating test cases, whereas Crawljax uses HTML and Morpheus uses XHTML JSF templates. These artifacts are indirectly associated to the widgets used in the UI of web applications and the user interaction scenarios. In order to increase the coverage of the different layers, one possible scenarios is analysing test-case generation strategies for the different layers.

Our approach is a black box test case generation approach in which test case generation is performed without having limited access to software artifacts (we only evaluate XHTML template models). The goal of our tool is to work with legacy systems (obsolescent platforms that have been in use within a company for many years), assisting their refactor activities, even when no automated test cases were implemented for them.

The system (Exactus CRM) in which the tests were performed is a small size system with approximately (11Kloc), with this, the generation of test cases was performed in around 1 minute. Generating test cases for other applications would not be difficult, the execution as it is a selenium script could take time. However there are options for accelerating the execution time of these scripts, such as running Selenium distributed in a grid to try to optimize the execution.

Finally, The time for executing both approaches was not analysed in our study. However, the generation of the test cases for Morpheus take (around 1 minute), considering the number of artifacts that composed our project. The execution time, on the other hand, averaged 40 minutes for both approaches, considering they run using the Selenium WebDriver API, similarly to System/Acceptance test cases.

6 Conclusions

This paper presented a process for generating functional test cases for web applications, that use complex interface components in the process of generating test cases. Our approach was implemented in a tool, called Morpheus Web Testing, and it was developed for identifying interface components in JSF/Primefaces templates to automatically generate test cases. In terms of the scope of the proposal, Morpheus Web Testing can be applied to any JSF project that uses the Primefaces framework for interface components.

We evaluated our approach in a case study, in an industrial setting, with a production JSF/Primefaces web application. The results indicate that there is evidence that Morpheus Web Testing achieves greater code coverage compared to state-of-the-art technique, because for all scenarios the Morpheus Web Testing achieved better coverage on average. Although the evaluation provided evidence that the approach outperforms state-of-the-art, more assessments need to be carried out to generalize our findings.

The contributions of this work were:

- Implementing a test case generation strategy which identifies complex interface components (widgets) and use their information to generate more elaborate test cases for web applications;
- Reducing costs associated to the testing process, in web applications that use the JSF frameworks and Primefaces, specifically considering the industrial setting in which Morpheus Web Testing was used.

The suggestions for future work are:

- Extend the case study to other web applications to possibly further generalize the findings of this paper;
- Extend the case study to perform oracle generation in order to identify the number of flaws identified by the approaches; and
- Morpheus Web Testing focus on JSF/Primefaces web applications, hence its usage is limited in comparisson to CrawlJax which can be used to any web application. Regardless, our approach for generating test

cases from higher abstraction level definitions, such as widgets, could be extended to other Widget libraries, such as Angular,[4] React,[5] jQueryUI,[6] among others.

The limitations of this work are:

- The case study was conducted using only one application. WUI that uses the JSF and Primefaces frameworks, in this case the Exactus CRM with 11KLOC;
- The case study was conducted using only one tool of functional test case generation such as the CrawlJax;
- The usefulness of the proposed tool is evaluated only in terms of the coverage achieved by the generated test cases (fault detection capability, for example, is not considered);and
- In the Morpheus Web Testing approach, were not exercised all the possibilities of interface components of framework Primefaces.

References

[1] Pekka Aho, Nadja Menz, Tomi Räty, and Ina Schieferdecker. Automated java gui modeling for model-based testing purposes. In *Information technology: New generations (itng), 2011 eighth international conference on*, pages 268–273. IEEE, 2011.

[2] Robert V Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[3] J Craig Cleaveland. Building application generators. *IEEE software*, 5(4):25–33, 1988.

[4] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. Model-driven software product lines. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 126–127. ACM, 2005.

[5] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. Webmate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, pages 11–15. ACM, 2012.

[6] Valentin Dallmeier, Bernd Pohl, Martin Burger, Michael Mirold, and Andreas Zeller. Webmate: Web application test generation in the real

[4] http://www.angular.io

[5] https://reactjs.org/

[6] https://jqueryui.com/

world. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 413–418. IEEE, 2014.

[7] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mário Jino. *Introdução ao teste de software*. 2007.

[8] Kit Edward. Integrated, effective test design and automation. *Software Development*, 21(2):36–38, 1999.

[9] Mark Fewster et al. Common mistakes in test automation. In *Proceedings of Fall Test Automation Conference*, 2001.

[10] Zahra Abdulkarim Hamza and Mustafa Hammad. Web and mobile applications' testing using black and white box approaches. 2019.

[11] Elisabeth Hendrickson. The differences between test automation success and failure. *Proceedings of STAR West*, 1998.

[12] Cem Kaner. Improving the maintainability of automated test suites. In *International Software Quality Week*, 1997.

[13] Ajeet Kumar and Sajal Saxena. Data driven testing framework using selenium webdriver. *International Journal of Computer Applications*, 118(18), 2015.

[14] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Automatic testing of gui-based applications. *Software Testing, Verification and Reliability*, 24(5):341–366, 2014.

[15] Atif M Memon et al. *Comprehensive Framework for Testing Graphical User Interfaces*. University of Pittsburgh Pittsburgh, 2001.

[16] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. The art of software testing, 2004.

[17] Bao N Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105, 2014.

[18] Roger S Pressman. *Engenharia de software*, volume 6. Makron books São Paulo, 1995.

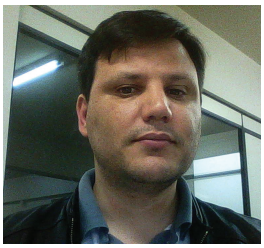
[19] Abdul Rauf, Sajid Anwar, M Arfan Jaffer, and Arshad Ali Shahid. Automated gui test coverage analysis using ga. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 1057–1062. IEEE, 2010.

[20] Ana Regina Cavalcanti da Rocha, José Carlos Maldonado, Kival Chaves Weber, et al. Qualidade de software: teoria e prática. *São Paulo: Prenttice Hall*, 2001.

[21] Marton Sakal. Gui vs. wui through the prism of characteristics and postures. *Management*, 5(1):003–006, 2010.

[22] Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.
[23] EDC Van Eyk, WJ Van Leeuwen, Martha A Larson, and Felienne Hermans. *Performance of near-duplicate detection algorithms for Crawljax*. PhD thesis, Citeseer, 2014.
[24] Tanapuch Wanwarang, Nataniel P Borges Jr, Leon Bettscheider, and Andreas Zeller. Testing apps with real-world inputs. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, pages 1–10, 2020.

Biographies



Romulo de Almeida Neves. Agile, eXtreme Programming, TDD, Hexagonal Architecture, and Flutter enthusiast. Graduated in computer engineering and master in computing. Over 17 years of experience with the Java platform, applicatons servers, elaboration of architectures, back-end projects, web solutions, desktop, mobile, solution integrations using rest, soap, microservices, using proprietary java products, JCP products, Jakarta EE and Spring framework products.

Willian Massami Watanbe. Ex-Yahoo!, Professor and Passionate Software Engineer, with experience in development practices associated to Web

technologies and Web engineering practices (such as: eXtreme Programming, TDD – Test-Driven Design, Continuous Integration and Continuous Delivery). Eager to contribute by developing quality assured Web applications, also considering attributes such as usability, maintenance and multi-platform characteristic of the Web.

Rafael Oliveira. Is a researcher in Software Engineering and adjunct professor – The Federal University of Technology – Paraná – UTFPR. Interested in research activities associated with planning/implementing automated software testing solutions in different projects, supporting code review processes, measuring change impact, planning and implementing testing frameworks, maintaining testing scripts, and reports, supporting API, and manual testing activities.