FEDERAL UNIVERSITY OF TECHNOLOGY - PARANÁ
OFFICE OF RESEARCH AND GRADUATE STUDIES
CORNÉLIO PROCÓPIO CAMPUS
GRADUATE PROGRAM IN INFORMATICS

GUILHERME RICKEN MATTIELLO

LEVERAGING AUTOMATED WEB TESTS INTO MODEL-BASED TESTING

MASTER THESIS

CORNÉLIO PROCÓPIO

2020

GUILHERME RICKEN MATTIELLO

LEVERAGING AUTOMATED WEB TESTS INTO MODEL-BASED TESTING

Thesis presented to the Graduate Program in Informatics of the Federal University of Technology - Paraná (UTFPR) as a partial requirement to obtain the degree of “Master in Informatics”.

Advisor: Prof. Dr. André Takeshi Endo

CORNÉLIO PROCÓPIO

2020

Dados Internacionais de Catalogação na Publicação

M444 Mattiello, Guilherme Ricken

Leveraging automated web tests into model-based testing / Guilherme Ricken Mattiello. – 2020.
58 f. : il. color. ; 31 cm.

Orientador: André Takeshi Endo.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Informática. Cornélio Procópio, 2020.
Bibliografia: p. 53-57.

1. Software - Testes. 2. Interface gráfica com o usuário (Sistemas de computação). 3. Banco de dados orientado a objetos. 4. Informática – Dissertações. I. Endo, André Takeshi, orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Informática. III. Título.

CDD (22. ed.) 004

Biblioteca da UTFPR - Câmpus Cornélio Procópio
Bibliotecário/Documentalista responsável:
Romeu Righetti de Araujo – CRB-9/1676

ACKNOWLEDGMENTS

First of all, I thank my family: my parents Névio and Zelia, my brother Eduardo and my fiancee Stephanie, who always encouraged me and were present in difficult times, understanding the moments that I had to be absent, because without the support and encouragement received it would be very difficult to reach the end of this journey. I dedicate this work to you!

To my advisor Prof. Dr. Andre Takeshi Endo, for the friendship, patience and dedication in advising this work and mainly for sharing his knowledge and guiding my steps, making the way easier, which was essential for the elaboration of this master thesis.

To my coworkers and all my friends who were present and accompanied me daily, reducing tension and directly influencing my academic training. I am especially grateful to my friend Henrique, for the exchange of knowledge and partnerships throughout this journey.

I would also like to register my gratitude to UTFPR professors, especially Érica and Watanabe, who made valuable contributions to my work, to the examining committee and to all the University staff for the support and knowledge received.

Finally, to everyone who, at some point, contributed to the realization of this work.

ABSTRACT

MATTIELLO, Guilherme. LEVERAGING AUTOMATED WEB TESTS INTO MODEL-BASED TESTING. 59 f. Master Thesis – Graduate Program in Informatics, Federal University of Technology - Paraná. Cornélio Procópio, 2020.

Background: Agile methods have driven automated execution of test cases, which have been adopted at different test levels, from unit testing to GUI testing. However, the tools that support automated testing focus on execution, leaving the generation of test cases as a manual task. In this way, the model-based testing (MBT) approach studies the generation of automated test cases through models, which are used to derive the test cases in a top-down workflow, where the model is created and from it are extracted the test cases that are subsequently executed. However, since testing implementation is currently the developer's own responsibility, the tester may come across a scenario where there is already a test suite that must be reused to produce new model-based tests. **Objective:** This work aims to present an approach that uses existing automated tests to facilitate the adoption of MBT in this scenario, allowing the reuse of GUI tests to derive new test cases. To support the evaluation of the approach, we developed a tool that uses the PageObjects pattern for the abstraction and structuring of existing test cases in event-based models. **Method:** The tool developed to evaluate the approach, called MoLeWe, supports the three steps of the approach: model inference, model extension and test case generation. The experiment was conducted with 18 students, who developed test cases in Java with the PageObjects pattern for 9 web applications. The tool generated models for the developed test projects, which were extended and used to generate new test cases. **Results**: The experimental study collected data such as line coverage of the new test cases, execution time, effort spent to generate new tests and information that gave evidence of the feasibility of the approach and reuse of the existing test suites. On average, line coverage increased by 38.97% with the new test cases, the execution time gave evidence of linear growth in relation to the size of the projects and the approach proved feasible, reusing most of the existing test suites, since, on average, 70.54% of the new events created were already concretized.

Keywords: Model-Based Testing, Automated Tests, Test Case Generation, PageObjects

RESUMO

MATTIELLO, Guilherme. LEVERAGING AUTOMATED WEB TESTS INTO MODEL-BASED TESTING. 59 f. Dissertação – Programa de Pós Graduação em Informática, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2020.

Contexto: Os métodos ágeis têm impulsionado práticas de execução automática de casos de teste, as quais têm sido adotadas em diferentes níveis de teste, desde testes de unidade até os testes de sistemas em interface gráfica (GUI). Porém, as ferramentas que apoiam o teste automatizado focam na execução, deixando a geração dos casos testes como uma tarefa manual. Desta forma, a abordagem de Teste Baseado em Modelo (TBM) estuda a geração de casos de teste automatizada por meio de modelos, os quais são utilizados para derivar os casos de teste em um fluxo *top-down*, onde o modelo é criado e dele são extraídos os casos de teste que posteriormente são executados. Entretanto, considerando que a implementação de testes é, atualmente, responsabilidade do próprio desenvolvedor, o testador pode se deparar com um cenário onde já exista uma suíte de testes que deva ser reutilizada para produzir novos testes baseados em modelo. **Objetivo:** Este trabalho tem como objetivo apresentar uma abordagem que utiliza testes automatizados existentes para facilitar a adoção de TBM neste cenário, permitindo a reutilização de testes GUI para derivar novos casos de teste. Para apoiar a avaliação da abordagem foi desenvolvida uma ferramenta que utiliza o padrão PageObjects para a abstração e estruturação dos casos de teste existentes, e modelos baseados em eventos. **Método:** A ferramenta desenvolvida para avaliar a abordagem, chamada MoLeWe, suporta as três etapas da abordagem: inferência do modelo, extensão do modelo e geração dos casos de teste. O experimento conduzido contou com a participação de 18 estudantes, os quais desenvolveram casos de teste em Java com o padrão PageObjects, para 9 aplicações *web*. A ferramenta gerou modelos para os projetos de teste desenvolvidos, os quais foram estendidos e utilizados para gerar novos casos de teste. **Resultados:** O estudo experimental coletou dados como cobertura de linha dos novos casos de teste gerados, tempo de execução, esforço gasto para gerar novos casos de teste e informações que atestaram a viabilidade da abordagem e reúso das suítes de teste existentes. Em média, a cobertura de linha aumentou 38,97% com os novos casos de teste, os tempos de execução de inferência do modelo e geração dos casos de teste forneceram evidências de crescimento linear em relação ao tamanho dos projetos e a abordagem se mostrou viável, reaproveitando grande parte das suítes de teste existentes, visto que, em média, 70,54% dos novos eventos criados já estavam concretizados.

Palavras-chave: Teste Baseado em Modelo, Testes Automatizados, Geração de Casos de Teste, PageObjects

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CES	Complete Event Sequence
ESG	Event Sequence Graph
FSM	Finite State Machine
GUI	Graphical User Interface
HTML	HyperText Markup Language
KLoC	Kilo Lines of Code
LoC	Lines of Code
MBT	Model-Based Testing
SUT	System Under Test
URL	Uniform Resource Locator
XML	Extensible Markup Language

SUMMARY

1 INTRODUCTION

Currently, it is common for systems to be developed or migrated to the web platform. Mesbah (2016) considers the web to be one of the most fascinating inventions of our time, for its great impact on society, as well as for delivering benefits such as inexistent installation costs, automatic updates for all users and universal access from anywhere in the world. With the growth of web, it is necessary to ensure that the systems developed on this platform have quality. This has attracted many Software Engineering researchers to the testing area, obtaining important advances in delivering a quality product on a web platform.

Testing is an activity focused on evaluating and improving the quality of a product, by detecting defects and problems (UTTING; LEGEARD, 2006). Software testing has become an essential practice and in recent years it has been one of the most researched topics in Software Engineering. This may be justified by the fact that software testing is often responsible for more than 50% of the software development cost (MYERS et al., 2011). According to (MAYES, 2010 apud GAROUSI; ELBERZHAGER, 2017), the costs of testing activities worldwide were 79 billion euros in 2010 and it was expected to increase to 100 billion in 2014. Therefore, it is important to improve techniques and approaches, automating them, and reduce costs involved in the test activity (ANAND et al., 2013).

Software testing has always been affected by the industry's growing demand for cost savings. However, extinguishing the test activity is not an option, always requiring faster and better tests, improving the goals, making them repeatable and transparent, as practiced in the Model-Based Testing (MBT) (KRAMER; LEGEARD, 2016). MBT is an approach that seeks to represent the behavior of a system being tested (*System Under Test* - SUT) through a model. This approach allows to generate test cases from the model to validate requirements and find bugs in the SUT.

In automated Graphical User Interface (GUI) tests of web applications, the Selenium WebDriver[1] framework is a reference in the creation of tests in different programming languages and for different browsers. A pattern that has been widely used with Selenium is PageObjects. This pattern reduces code duplication, facilitates maintenance and increases code reuse, abstracting interactions with web pages in a class called PageObject (SAMS, 2015).

[1]https://www.selenium.dev/

1.1 MOTIVATION

It is common for teams to have automated test suites as a way to reduce the testing costs. Other option is automatic test generation, which has some limitations, such as the generation of oracles (BARR et al., 2015); that can be mitigated by taking advantage of the knowledge embedded into an existing test suite implemented by a developer in a programming language (which we refer to as user-defined test suites). In this context, Adamsen et al. (2015) combined the two techniques (user-defined and automatic test generation) in mobile applications. They took advantage of user-defined tests by automatically injecting neutral event sequences that exposes the test case to adverse conditions. The user-defined test cases helped mainly in the assertions of the automated tests, since they contained relevant information about the problem domain.

An approach that supports the automatic generation of test cases is MBT, which uses models to automatically derive test cases, taking advantage of the domain knowledge injected into the models. MBT is interesting because, in addition to requirements validation, it increases the rate of defect detection, reduces the time and cost of testing activity, increases the quality of tests, and helps in the tests traceability (UTTING; LEGEARD, 2006). Many tools support the MBT approach in different stages, from creating models to generating concrete and executable test cases in the SUT. Farto and Endo (2017) presented, through the `FourMa` tool, an approach to reuse test models of mobile applications with the objective of reducing the effort spent in the activity of generating test cases.

However, the MBT approach is used only in some specific domains in the industry. Much is due to the infeasibility of the solutions implemented by researchers who do not take into account the cost of implementation (GAROUSI; FELDERER, 2017). In addition, many projects in industry already have a test suite, so it would only be desirable to use MBT if existing tests were to be taken in account.

In their studies, Kramer and Legeard (2016) found that the initial effort to create test cases with MBT is greater than that of traditional approaches. However, after generating the model, maintenance becomes faster and more efficient, achieving a cost reduction of approximately 75% in the maintenance phase. Thus, suppressing the initial cost of generating the model, like generating it in a semi-automated way, the MBT approach may stand out over other traditional approaches. Knowledge of the problem domain in user-defined test suites and the agility and regularity of automated tests are important in the construction of the tests. In this way, it is interesting to extract the model of an SUT from existing test suites.

While MBT has been performed in organizations that adopt agile methods (KRAMER et al., 2017), there is a lack of studies that investigate how MBT can be effectively applied in such contexts. For instance, agile methods and their practices have fostered the presence of automated test cases. Such tests have been successfully and extensively adopted to verify different software levels, from unit tests (e.g., JUnit) to end-to-end Graphical User Interface (GUI) tests (e.g., Selenium WebDriver). They are more prevalent since writing automated tests is nowadays a developer's responsibility, as advocated by agile practices like test-driven development (TDD) (BECK, 2003) and behavior-driven development (BDD) (CHELIMSKY et al., 2010), resulting in suites with a large amount of tests loaded with knowledge about expected input and output data, through oracles (DANGLOT et al., 2019a). In these circumstances, testers may encounter scenarios where a test suite, implemented by developers, may be leveraged to derive new test cases from it.

Even though there exists an effort of the community to leverage existing test suites in order to improve the test cost and effectiveness (XIE; NOTKIN, 2006; FRASER; ZELLER, 2011; ZHANG; ELBAUM, 2014; FARD et al., 2014; ADAMSEN et al., 2015), the literature lacks an in-depth investigation of how MBT can be integrated in such a context. Existing approaches have been applied in scenarios where test cases are specified in textual formats (TORENS et al., 2011b; SCHULZE et al., 2015; DIXIT et al., 2015), and automated tests written in programming languages like Java have not been taken into account. We surmise that if MBT could be incrementally integrated in a context where automated tests are already used, the resistance would be smaller and MBT could be more widely adopted.

1.2 OBJECTIVES

Nowadays, there is a need to find solutions that take advantage of test cases written in programming languages like Java and follow the development methodologies and techniques currently practiced. This work aims to present and evaluate an approach to improve the synergy between existing test cases (coded in a programming language) and model-based testing. With focus on web applications, end-to-end GUI automated tests, and an event-driven technique to represent the test models, this approach involves *(i)* model inference from a source code repository with automated tests, *(ii)* test extensions at model level, and *(iii)* abstract and concrete generation of new tests from the extended model. To check the feasibility, a prototype tool was developed to support the proposed approach, and an experimental evaluation was conducted with nine open source web applications and 18 participants.

1.3 TEXT STRUCTURE

This work is organized as follows: Chapter 2 presents the concepts of automated tests, model-based testing, the technologies and patterns involved in the development of the work. Related works are also discussed. Chapter 3 introduces the proposed approach, detailing each of the steps involved, and presents the prototype tool used to support the approach. Chapter 4 presents the evaluation of the proposed approach. The procedures involved in the experimental study are detailed and the results are discussed. Finally, in Chapter 5, the contributions of this thesis and future works are presented.

2 BACKGROUND

This chapter presents the concepts necessary to understand the work. Section 2.1 introduces MBT and discusses its fundamental aspects, as well as presenting the Event Sequence Graph modeling technique. Section 2.2 addresses the topic of automated testing, with an emphasis on GUI testing for web applications. It also introduces the Selenium WebDriver framework and the PageObjects pattern. Finally, Section 2.3 discusses related works.

2.1 MODEL-BASED TESTING

According to Myers et al. (2011), testing is a process that seeks to ensure that software correctly does what it should be doing, comparing actual and expected behavior and, thus, detecting defects. However, as it is not possible to test all combinations of software input and output, it is necessary to apply testing techniques that optimize this activity and guarantee the quality of the SUT.

Utting and Legeard (2006) define MBT as the automation of the black box test design, with the advantage that the model created to represent the behavior of the SUT captures the requirements as well, allowing to validate them during the process. With that, MBT tools are used to generate the test cases automatically from the model.

Among the advantages of MBT are the improvement in communication between testers and stakeholders, the definition of a unique understanding of the requirements and the automated generation of test cases from the model, which is a way of managing knowledge (KRAMER; LEGEARD, 2016). In addition, the literature also found that MBT provides a high error detection rate, reduces the cost and time of testing, and facilitates traceability (UTTING; LEGEARD, 2006).

Kramer et al. (2017) identified, through a survey, that the use of MBT for system-level testing increased in the period from 2014 (49.5% of respondents applied MBT in system-level testing) to 2016 (77.6% of the interviewees applied MBT in tests at the system level) and that in 2016 the models became simpler, containing less details, compared to 2014. The authors also observed that from 2014 to 2016 the number of MBT supporters grew in companies that follow agile methodologies.

Models are detailed representations of systems at smaller scales, but which represent all the characteristics of the system at a relatively low cost. In the case of MBT, the model represents the behavior of the system in inputs and outputs, so that the output of the model is equivalent to that of the SUT. This allows structuring the tests, which are often irreproducible because they are not documented (UTTING; LEGEARD, 2006).

MBT can be divided into four main stages (ENDO, 2013), represented in Figure 2.1 and described below.

1. Modeling: It is necessary to know the requirements of a system and its functionalities, as well as characteristics of the environment in which the system is hosted, such as operational system, applications and libraries. In this step, a model of the SUT will be created, which must be smaller and simpler than the SUT, but which must contain details that accurately describe all the parts of the software to be tested.

2. Test case generation: This step depends on the modeling technique that describes the test models, since from these models test cases will be generated. The results of this step are sequences of events extracted from the model.

3. Concretization: The test cases generated in the generation stage will be transformed in order to be executed in the SUT, aiming to automate the entire process. This transformation or concretization of the test cases is done using transformation tools that use various templates or mappings to translate the test cases into executable codes, or using adapter code. Adapters are codes that allow abstract inputs to be run in an SUT. They concretize and execute them to compare the expected result with the output result of the SUT.

4. Test execution: In this step, the test cases, after they are concretized, will be executed in the SUT.

Despite academic efforts to improve testing techniques, Garousi and Felderer (2017) show that topics of interest in industry and academia are different, as what is important to industry is not so important to academia and vice versa. For example, academic studies have focused on challenges related to theoretical problems (such as combinatorial testing and search-based testing), while the industry has sought to improve the efficiency and effectiveness of tests using simple methods.

MBT is popular in academia, but it has limitations when used in practice. Many practitioners believe that academic papers are very formal, difficult to understand and do not follow cost-benefit analyses of the proposed solution, making the application unfeasible and taking the industry's interest out of it (GAROUSI; FELDERER, 2017). Thus, there is an opportunity to create MBT solutions that are simpler and feasible in practice. As an example, it is interesting that an approach takes advantage of existing test cases in common scenarios, so that it

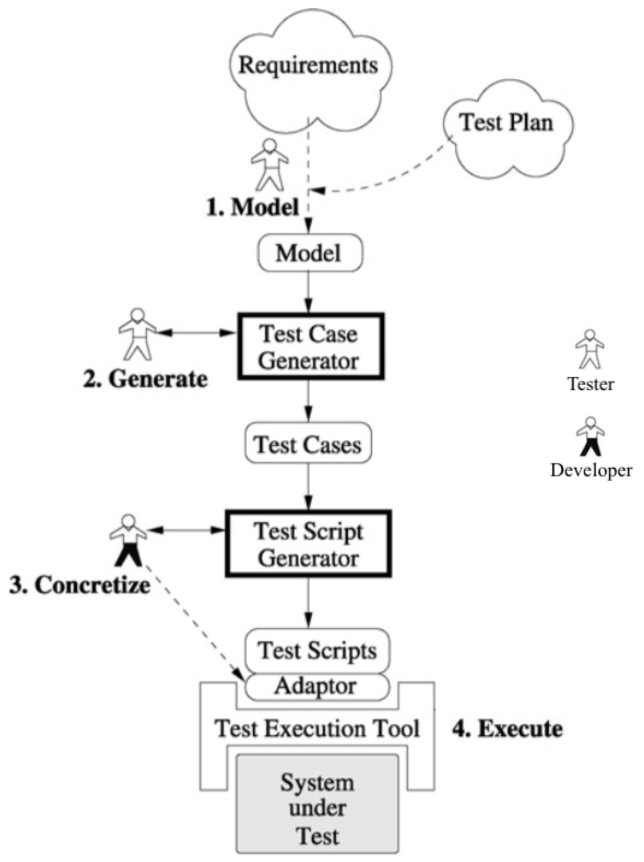


Figure 2.1: MBT steps. Adapted from (UTTING; LEGEARD, 2006)

can be applied to teams that use agile development techniques, where the programmers themselves implement the tests. In addition, it is important to consider compatibility with continuous integration tools.

2.1.1 EVENT SEQUENCE GRAPH

For the construction of an MBT model it is necessary to choose a modeling technique, and several techniques can be used, for example, UML diagrams (state machine diagram and activity diagram). Among the techniques, this work focuses on Event Sequence Graph (ESG), a technique used to model possible interactions between a user and a system, through events. An event is a phenomenon that can be triggered by a user stimulus or as a system response, representing different states of the system's activities (BUDNIK, 2006).

An ESG is a directed graph, where the user's events and interactions with the system are represented by vertices (or nodes) and the valid sequence of execution of these events is defined by the edges that connect them. In its notation, the initial node is defined by the pseudo vertex symbolized by "[" and the end is defined by the pseudo vertex "]". A sequence of the ESG that starts at the initial vertex and ends at the final vertex is called Complete Event Sequence (CES). Figure 2.2 illustrates the ESG that models the Google search functionality.

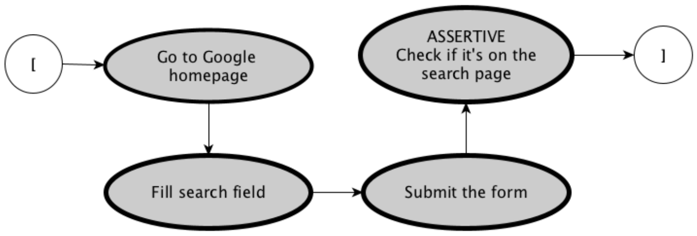


Figure 2.2: ESG of Google search functionality.

Web applications can be viewed as event-driven systems, so using ESGs becomes an advantage, allowing to model not only the expected behavior of a system, but also the unexpected interactions between the user and the SUT. ESG is widely applied for using formal notations of consolidated theories, such as graph theory and automata theory; another advantage is that can be learned in a short period of time and supported by specific tools (BELLI et al., 2014).

2.2 AUTOMATED TESTS

Even though it is a high-cost activity that requires intense work, software testing is indispensable for software development. This activity is responsible for more than 50% of the total development cost, so it is important to reduce costs and improve the effectiveness of tests through automated tests (ANAND et al., 2013). Automating the tests brings several benefits regarding reuse, repeatability, code coverage and less effort in executing them, which allows even greater advantages in regression test suites (RAFI et al., 2012).

Despite the visible advantages of adopting automated tests, this area of study is not yet fully explored and understood. Wiklund et al. (2017) conducted a systematic review in order to identify difficulties in the automated testing activity. Some points raised by the authors should be considered when adopting automated tests:

- The learning curve for automated tests is high and often the complexity of the tools that assist this task contributes to this. This supports the importance of choosing a tool that does not require high skills from the team, so that the tests bring results with high cost benefit;
- Between 30% and 80% of software development costs are used for testing activities. The time and cost involved can be reduced by employing automated tests, as the tests are

easily reproducible, leaving the team available for other activities and receiving quick feedback from the SUT;

- The people expectations, resulting from this activity, are extremely high and this can hinder the progress of the project if the expected results are not achieved in the desired time. Managing these expectations is an important task, because if short-term results disappoint stakeholders, the tests will be considered as waste;
- The cost of a software project often exceeds the planned budget, becoming a limiting factor for testing automation. Therefore, there are examples of organizations that consider some projects too small for efficient testing automation;
- Software testing is part of software development and should be considered a valuable and indispensable activity in the development process. If the team does not have a well-defined methodology, which includes the testing activity, this step will be neglected and the organization will see no value in automating the tests;
- There are also difficulties inherent to the investment and incentive of stakeholders. Some systems have a low testability, for example, GUI implemented in a way that does not facilitate or allow machine-to-machine interaction, making it difficult to carry out automated tests with the available tools. In these cases, it may be more effective to improve the testability of the SUT, rather than keeping test cases functional for the GUI in question.

Among the disadvantages are also a high initial cost in the test cases generation, choosing accessible tools and training. Rafi et al. (2012) also found, by means of a survey, that 45% of the respondents recognize that there is a lack of tools that fully meet their needs, mainly in relation to the learning curve and maintainability of the test cases that the tool provides. In addition, 80% of respondents believe that automated testing will not replace manual testing completely.

Garousi and Elberzhager (2017) divide the test activity into six categories that can be automated: creating test cases, generating test codes, executing tests, evaluating tests, reporting results and managing test activity. In this work, the focus is on generating test cases and creating test codes from models, more specifically for GUIs. In this scenario, the record-replay tools are widely used because they are practical and simple to use. However, these tools give the tester less control over the test suite and generate code that is considered fragile, as any minimal change to the GUI can break the generated code (GAROUSI; ELBERZHAGER, 2017; ENTIN et al., 2011).

In general, test automation lacks practices and tools that facilitate its adoption. For example, when developing a tool, simplicity and the learning curve it requires from users must be considered. In addition, the tool must adapt to the development methodology or establish well-defined steps and flows that avoid neglect when implementing the tests. Regarding the methodologies, it is important to consider the strong relation that currently exists between agile methodologies with the use of integration and continuous delivery tools, increasing the quality of the test activity at a reasonable cost (GAROUSI; ELBERZHAGER, 2017). Thus, proposed tools must also be adaptable to this scenario.

2.2.1 SELENIUM WEBDRIVER AND PAGEOBJECTS

One of the most used tools in automated tests for systems on the web platform is the Selenium WebDriver (LEOTTA et al., 2013). This tool allows practitioners to run tests on the most popular browsers, such as Google Chrome, Mozilla Firefox and Safari, through its own drivers. Tests can be written in programming languages like Java, C#, Ruby and Python.

Figure 2.3 illustrates a simple test case written in Java using the Selenium WebDriver. Line 4 tells the browser which Uniform Resource Locator (URL) to access for testing. Line 5 verifies, by means of an assertion, if the browser is in the correct state, on the Google homepage. Line 6 searches for the element of the web page corresponding to the search field, which represents an HyperText Markup Language (HTML) element locator by the name attribute. Line 7 inserts the value “Selenium Essentials” in the search field and line 8 submits the form.

```
1 @Test
2 public void Selenium_Essentials() throws Exception {
3      // Make the browser get the page and check its title
4      driver.get("http://www.google.com");
5      Assert.assertEquals("Google", driver.getTitle());
6      WebElement element = driver.findElement(By.name("q"));
7      element.sendKeys("Selenium Essentials");
8      element.submit();
9 }
```

Figure 2.3: Selenium test case example (SAMS, 2015)

PageObjects is a test design pattern that has been widely used in conjunction with the Selenium WebDriver framework, as it defines a web page using objects. This page is abstracted into a object-oriented paradigm class where the attributes are usually elements of the web page and the methods are actions and events that manipulate the elements of the page (SAMS, 2015).

According to Fowler (2013), the basic rule of the PageObject classes is to allow an automated test to see everything that a human being can see. For example, to access a text field, the class must provide methods that insert and return a string. In this work, this characteristic of the methods being intelligible by machines and humans will facilitate the mapping of the generated model to the source code. In addition, the use of PageObjects prevents the test code from being vulnerable to changes in the GUI, as it concentrates the HTML element locators in the PageObjects classes. The use of PageObjects also helps in the evolution of the web application, as it generates more robust codes and promotes reuse, readability, maintainability and the low coupling between the test code and the web application (STOCCO et al., 2015).

Leotta et al. (2013) investigated the advantages of using the PageObjects pattern in software maintainability, comparing a suite implemented with PageObjects with a suite without them. In a case study, after performing a software update, they concluded that the suite that followed the PageObjects pattern took less time to be corrected and fewer lines of code were changed to suit the software modifications. Christophe et al. (2014) analyzed the quality of web application projects that have Selenium test suites. Most of the projects categorized as high quality use frameworks that implement the PageObjects pattern.

Figure 2.4 illustrates the test case in Figure 2.3 refactored to use PageObjects. Lines 1-18 represent the PageObjects class that corresponds to the Google homepage. Lines 22-27 implement the test case using the PageObjects class. Lines 3-4 specify how to locate the Google search field using the name attribute. The constructor method on lines 8-11 informs the URL that the browser must access. Lines 13-16 represent the search method, where the parameter passed to the function will be inserted in the search input of the web page and then the form is sent (line 15). In lines 24-25, the test case invokes the constructor of the PageObjects class, so that the browser shows the Google homepage and in line 26 the search is performed with the value "Selenium Essentials".

Considering the approach proposed by this work, the use of this pattern will allow the model generated to be more readable and organized, promoting the reuse of test codes. At the model level, each event in the model will be mapped with one method of the PageObjects classes. Test cases are the sequences of possible paths that can be extracted from the model and executed through method calls from PageObjects.

2.3 RELATED WORK

Automated software testing has an extensive literature with several contributions from both industry and academia (ANAND et al., 2013). Among them, MBT has been investigated in

```java
public class GoogleSearchPage {

  @FindBy(name = "q")
  private WebElement searchme;

  public WebDriver driver;

  public GoogleSearchPage(WebDriver driver) {
    this.driver = driver;
    driver.get("http://www.google.com/");
  }

  public void searchFor(String text) {
    searchme.sendKeys(text);
    submitme.submit();
  }

}

...

@Test
public void Selenium_Essentials() throws Exception {
     GoogleSearchPage page =
         PageFactory.initElements(driver, GoogleSearchPage.class);
     page.searchFor("Selenium Essentials");
}
```

Figure 2.4: Test case example using Selenium with the PageObjects pattern - Adapted from (SAMS, 2015)

order to formalize and automate test generation. While the community has reported successful cases of MBT adoption mostly in critical contexts like embedded systems (ZANDER et al., 2011), (KRAMER et al., 2017), few studies have examined how MBT can be applied and systematically improve the effectiveness when there exist automated tests.

Amplifying test suites. Traditionally, the generation of test cases does not consider existing tests. However, with the rise of agile methodologies, which foster the implementation of tests frequently and in advance, a new research trend has emerged known as test amplification, which leverages manually written tests. In a survey, Danglot et al. (2019a) identified works that analyze and operate on existing test suites. The authors divided the work into four categories: (i) adding new tests; (ii) synthesize new tests based on changes in the code; (iii) modify the execution of the tests and (iv) modify existing tests. This work falls into category (i), as it seeks to amplify the test suite by adding new tests, based on the existing test codes. At first, changes to existing tests will not be allowed, as in category (iv), as it will be considered that these tests are valid and efficient, as these tests will be the basis for creating new tests. The following are some works that amplify tests and that are close to the focus of this work.

Farto and Endo (2017) used the `FourMa` tool to test mobile applications (Android) in which the MBT approach is applied. The objective was to reduce the effort required in the concretization of test cases and to test features that are normally ignored in manual tests, such as unexpected events. For this, the test model was reused and specific events, such as hardware, sensor events, unpredictable user interactions and telephone events, were inserted in the middle of the test cases.

There are authors who study amplification and refactoring of unit tests. Tillmann and Schulte (2006), Thummalapenta et al. (2011) and Fraser and Zeller (2011) investigate ways to refactor conventional unit test cases, transforming them into parameterized test cases, or generate test cases that are already parameterizable. Among the measures adopted, the main one consists in replacing concrete values of the body of the methods with parameters, through symbolic execution. This generalizes the test cases and brings benefits such as less computational effort, makes the test cases simpler and more expressive and increases the code coverage, since it is possible to cover other paths by running the test case with several test entries. Xie and Notkin (2006) extract operational abstractions from a unit test suite to guide test case generation tools, so that they can generate new data entries that violate those abstractions. The approach assumes that, by violating existing tests, a new functionality of the program will be covered, which was not covered by the existing test suite.

Most of the works surveyed by Danglot et al. (2019a) seek automated ways to generate new test cases, oracles or inputs to improve code coverage, defect detection, debugging or decrease the cost involved in generating the test cases. However, the authors concluded that there are still gaps in how to take advantage of the information contained in existing test cases. Many approaches generate large amounts of new test cases, which makes it difficult to manage the test suite, and it is not yet trivial to generate oracles for the generated tests. Therefore, it may be interesting to approach these problems manually, with the influence of developers or testers, providing greater control over the new test cases generated and injecting domain knowledge into the generated test cases.

Zhang and Elbaum (2014) propose mechanisms to amplify test cases in order to validate exception handling code. Danglot et al. (2019b) sought to automatically improve existing test cases by implementing a tool called DSpot that: receives test cases in JUnit, applies improvements, and returns to developers as pull requests. Improvements are inserted as changes to values and objects, adding new method calls and assertions. The authors conducted a quantitative study with 40 real-world test classes from 10 open-source Java projects. As outcomes, the authors got 13 pull requests accepted and the tool was able to improve the initial mutation score of a test class by 99%.

There has also been some work on system-level GUI testing. Fard et al. (2014) use existing GUI tests for web applications. Their work proposes finding a solution, in these existing tests, to three main automated test limitations: automatic filling for input values in form fields, paths to explore on the SUT and test oracles. Using this knowledge injected in user-defined test suites, they developed a tool (`Testilizer`) that extends the test cases, in an automated way with a crawler, to parts not covered by the SUT. They observed an 150% improvement in fault detection and 30% in code coverage. The authors' proposal seeks to take advantage of assertions implemented manually to extend the model in an automated way, not allowing the tester to evolve the model manually, as in MBT. Furthermore, it differs from the present approach in that it uses a state flow graph to represent test cases at the model level, where the states are represented by the current DOM of the application and the transitions are actions.

Adamsen et al. (2015) describe an approach to test mobile apps by leveraging existing test suites to adverse conditions. They insert events, in a systematic way, that can interfere with its correct execution. The approach was implemented through the tool named `Thor`, which injects events that influence the state of an Android GUI and the audio service (such as loss of audio focus), and take advantage of user-defined test suites to verify if the SUT's behavior remains correct.

MBT in an agile context. Entin et al. presents contributions produced during years of application and improvement of MBT in a SCRUM project of the company OMICRON. With a focus on improving the accuracy and completeness of models created, the authors list as benefits the participation of the entire development team in the definition of the model in early stages. The authors have already validated the use of capture-replay tools in the generation of test cases and formal notations in the definition of requirements (such as Gherkin), which allowed less experienced testers to also produce parts of the model and decrease maintenance efforts on the model (ENTIN et al., 2011), (ENTIN et al., 2012), (ENTIN et al., 2015).

Li et al. (2016) investigated how to generate tests for the Cucumber tool in an automated way. As a result, they developed an MBT tool called Skyfire, which generates Cucumber-compatible tests from UML behavioral models (such as state machine diagrams). The Skyfire tool uses a tool that reads UML models and generates abstract test cases (from the inserted UML model). From that, Skyfire converts the abstract test cases into scenarios of the Cucumber tool. With the scenarios created, testers implement the methods mapped by Cucumber.

While these approaches aim to integrate their approach in an agile context, existing automated tests are not taken into account.

MBT and automated tests. Automated tests are studied constantly and there are numerous contributions in the literature. Among them, MBT has been widely applied to test case generation, mainly in domains of critical and embedded systems, but it still lacks studies applied to web systems (GAROUSI; FELDERER, 2017).

Some authors have applied the MBT approach in an inverse way, that is, generating a model from the test cases. Torens et al. (2011a) take advantage of abstract test cases, of a system with a railway domain, to create a model of the system's behavior and, later, use the model for prototyping. The inverse approach allows reusing existing test cases, so that domain experts can extend the model, creating test cases focused to the requirements and thus generating an initial prototype of the SUT's behavior. In addition, existing test cases can be validated.

Schulze et al. (2014) conducted an empirical study with the objective of comparing tests written manually, without automated steps, and tests written using the MBT approach, with automated generation and execution. Manual tests were conducted by a tester from a development company and tests carried out using the MBT approach were conducted by a tester from a research center. The authors concluded that manual testing required less preparation time, but the test cases did not fully cover the SUT. On the other hand, the tests that followed the MBT approach required more initial preparation time, were implemented more systematically and detected more defects. While tests carried out manually detected more inconsistencies in the GUI, such as incorrect labels, MBT identified more functional problems.

Mesbah (2016) analyzes the important achievements and advances in the area of software testing. Among them, the `Crawljax` tool, proposed by Mesbah et al. (2012), stands out. The `Crawljax` was proposed to extract a state flow graph from an SUT, exploring the changes in DOM resulting from events. This tool was created to explore web applications that make intense use of Javascript events to dynamically manipulate the DOM on the client side.

Dixit et al. (2015) use manually specified test cases to reverse engineer an Finite State Machine (FSM) of the web application. The authors assume that test cases are specified in an adapted Gherkin[1] input file. Their approach focuses on test initialization process and new tests are generated by exploring the extracted model. Torens et al. (2011b) used existing test cases, written in textual language, to generate a test model with the objective of modeling the behavior of the system and creating a functional prototype. Both works use test cases written in formal textual language and not in programming languages, such as Java, restricting the pragmatism of the approach.

[1]Gherkin is textual language similar to English to support the specification of scenarios in the Given-When-Then format.

Biagiola et al. (2017) automatically extracted navigation paths in an SUT, defined implicitly in the PageObjects classes developed by the testers, to generate test cases. The authors developed a tool called `Subweb` and compared the results with test cases generated using crawling techniques. They concluded that, although the proposed approach requires the manual writing of the PageObjects classes, the developed tool generates smaller navigation graphs in relation to those generated through crawlers, does not generate unfeasible (divergent) test paths and the coverage achieved by `Subweb` was 96% versus 83% for the other approach.

Torens et al. (2011b) report an experience on introducing MBT in a large scale industrial project. They investigate the context in which existing test cases are used as basis for model creation. The authors focus on textual test cases and the derived model is intended to be used for system modeling and prototyping. While they claim such derived model can be used to apply MBT and generate new tests, no further discussion is provided.

Schulze et al. (2015) describe an approach to extract a FSM model from Selenese[2] test cases; its model's transitions and states represent assertions and actions in the test cases. From this model, additional test cases are then generated with the tester's guidance. The authors evaluated the approach with an industrial case study in which new faults were detected. The authors assume that the test cases are in Selenese scripting language generated by the capture-replay Selenium IDE tool.

While test cases can be specified in textual formats (TORENS et al., 2011b; DIXIT et al., 2015) or Extensible Markup Language (XML) scripts (SCHULZE et al., 2015), nowadays most automated test suites are written in a programming language like Java. Therefore, our approach differs from these studies by reusing automated tests coded in a well-known programming language. Besides, such tests are maintained by developers and are expected to continue to exist even with the introduction of MBT.

Biagiola et al. (2019) implemented a tool (`DIG`) to extract a navigational model of the SUT, where the transition labels follow the PageObjects pattern, and derive new candidate test cases, using diversity-based heuristics, to maximize the navigation sequence and input data diversity. Then, `DIG` evaluate the quality of new test cases, using a novel metric distance between test actions and input data, without executing them, just comparing with previously test case candidates. They identified that `DIG` generates test suites with higher coverage and fault detection than crawling-based and search-based approaches.

Others. Micskei (2015, 2017) cataloged MBT tools that generate test cases from a model using various paradigms and techniques. Tools that generate new tests from existing

[2]XML-based language used to represent commands for Selenium.

test codes were also investigated. In none of the tools raised, the authors found the generation of a model from existing test cases with subsequent amplification of existing tests from the generated model.

Bernardino et al. (2017) sought, through a systematic review, to raise the tools and models used by works in the literature that apply the MBT approach. At the end of the review, 87 studies were selected and 70 tools to support MBT were identified. Among the works that contributed with a tool or tests aimed at web applications, no tools similar to the one proposed in this work were found. The tools and works found addressed, among other topics, regression tests using models based on XML Metadata Interchange (ZECH et al., 2012), security tests in web applications (XU et al., 2012), mutation tests at model level (AICHERNIG et al., 2015), web tool for the distributed creation of tests on modeled systems in finite state machines or state diagrams (ARANTES et al., 2014) and generation of test cases from models written in the state machine format (SLACK, 2011).

Tools from industry (e.g., Monkey,[3] Gremlins.js[4]) and from academia (e.g., Crawljax (MESBAH et al., 2012), ProCrawl (SCHUR et al., 2013), Sapienz (MAO et al., 2016), Stoat (SU et al., 2017)) have shown effectiveness to detect faults and increase code coverage (CHOUDHARY et al., 2015). These tools aim to explore automatically the GUI of apps, using black, white, or grey-box analysis techniques.

Unlike the aforementioned related works, we propose a practical approach that combines the advantages of MBT with frequent agile web development scenarios, where test cases are written in common programming languages like Java. Most of the listed approaches generate new test cases from models automatically, but we propose a manual evolution of the model, so that the generated test cases may be directed to cover the most critical parts of the SUT. Thus, we avoid common problems of automatic test generation, such as generating input values and assertions, and we minimize the initial effort to apply MBT, with automatic model inference. This allows us to take advantage of manual testing, such as injecting human knowledge into the test suite, and automated testing, through the reuse and generation of some code snippets automatically, so that the tester has greater control over the process and drives the continued growth of the test suite.

[3]http://developer.android.com/tools/help/monkey.html
[4]https://github.com/marmelab/gremlins.js

2.4 FINAL REMARKS

This chapter introduced the necessary concepts to understand this work. We also listed the related works, highlighting their similarities with the present work. The next chapter introduces the proposed approach, emphasizing its characteristics, explaining the steps involved and presenting the prototype tool implemented to evaluate it.

3 APPROACH TO INTEGRATE AUTOMATED TESTS AND MBT

This chapter introduces an approach that aims to improve the integration between existing test suites and MBT. The focus is on testing at system level since it is the context where MBT has been mainly applied (KRAMER et al., 2017). In particular, the approach is illustrated with GUI test cases for web applications. Also, we present the tool implemented to support the proposed approach. Section 3.1 introduces an overview of the approach. Sections 3.2, 3.3 and 3.4 explain each of the three steps of the approach: model inference, model extension and test cases generation, respectively. Finally, Section 3.5 presents the tool, providing a description of how it works and exemplifying its usage with PetClinic.

3.1 OVERVIEW

Figure 3.1 summarizes the proposed approach; its workflow contains three main steps. First, models are inferred from the test cases and made available to testers. Second, testers can extend the models to verify different aspects. From this, there are two possible directions for the third step: *(i)* some parts of the extended models have no automation code, so abstract test cases are generated to cover such parts and sent back to developers for implementation; and *(ii)* other parts are already automated, so concrete test cases are generated and executed. Each step is discussed in detail as follows along with a running example.

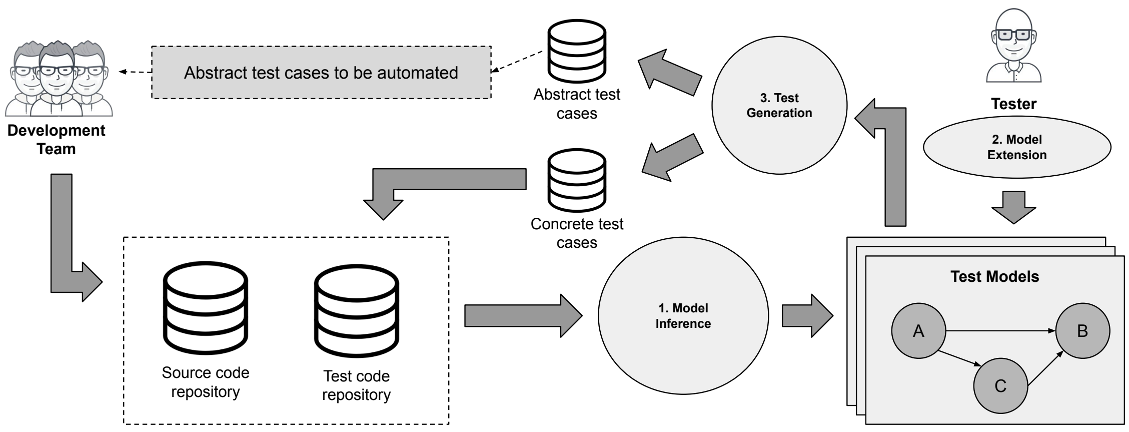
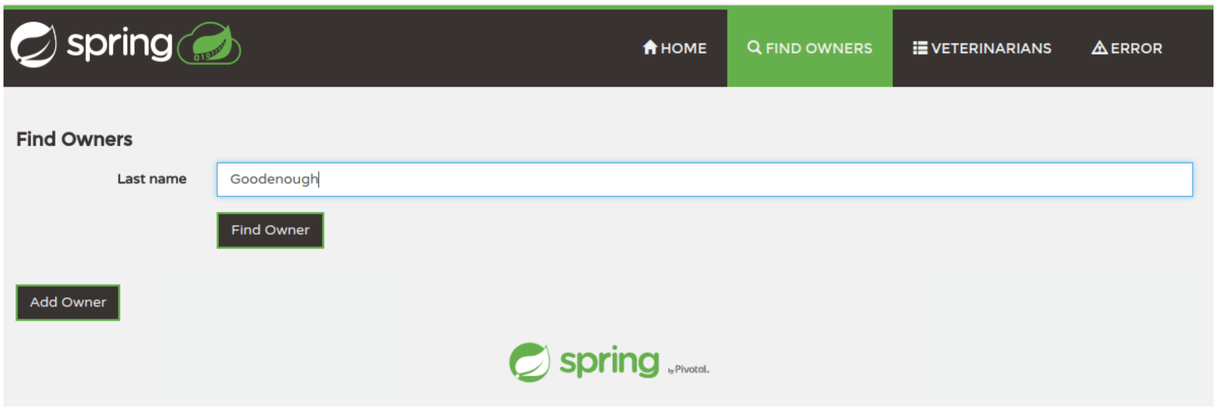


Figure 3.1: Overview of the approach.

To illustrate the proposed approach, we adopted PetClinic as running example. *PetClinic* is a sample Web application provided by Pivotal' Spring to show off the main features of its framework (SPRING.IO, 2014). It provides a set of features like list of veterinarians, management of pets and owners, and registration of visits. Figure 3.2 shows a screenshot of the find owners page. Figure 3.3 illustrates a typical test case implemented in Java using the HomePage and FindOwners classes as PageObjects. To reach the page in Figure 3.2, line 3 instantiates a new PageObject HomePage, lines 4-5 navigate to FindOwnersPage, in lines 6-7 the last name is set and the button find is clicked. Finally, line 8 asserts if the message of not found is displayed.



Find Owners

Last name: Goodenough

Figure 3.2: PetClinic Find Owners Page

```
1 @Test
2 public void testFindInexistentOwner() {
3   HomePage homePage = new HomePage(driver);
4   FindOwnerPage findOwnerPage = homePage.getMenu()
5     .goToFindOwners();
6   findOwnerPage.setLastName("Goodenough")
7     .clickFindButton();
8   assertEquals("has not been found", findOwnerPage.getErrorMessage());
9 }
```

Figure 3.3: Test case with PageObjects.

3.2 MODEL INFERENCE

This step receives as input the test cases coded using appropriate test abstractions (PageObjects) and provides means to infer test models. In this work, the model is represented as an ESG augmented with parameters to describe input data (this is further discussed in Section 3.5). We employ a simple traceability mechanism that maintains the consistency between the test code repository and the models; it has three main rules:

1. Each test case is represented in a test model, but the parts representing it cannot be modified at model level.

2. Each event (at model level) is mapped to one method call to PageObject classes or assertions.

3. The creation of new events (at model level) emplies in modifying PageObjects associated (at code level).

To attend Rule (1), Algorithm 1 describes a procedure to infer ESG-based test models from a set of test classes. From code to model perspective, the inferred models also respect Rule (2). Overall, a test model is inferred for the entire project (Lines 1-19). It iterates over the method call (event) sequence (Lines 5-15) for each method of all test classes and, *(i)* merges if the current prefix exists in the model and the last method call has parameters (Lines 6-8), or *(ii)* create a new event and connect it to its prefix (Lines 9-14). The algorithm is inspired by the prefix tree acceptor for regular languages (ANGLUIN, 1982).

Algorithm 1: Infering test models

```
input  : Test classes $TCl$
output: Test model $tm$
1  create new test model $tm([,])$
2  foreach $clazz \in TCl$ do
3      foreach $m \in methods(clazz)$ do
4          extract method calls $mc_1..mc_n$ from $m$
5          for $i \leftarrow 1$ to $n$ do
6              if $tm.hasEventSequence(mc_1..mc_i)$ then
7                  $e \leftarrow tm.getEventFromTailOf(mc_1..mc_i)$
8                  if $mc_i.hasParams()$ then $merge(e, mc_i)$;
9              else
10                 $e \leftarrow tm.createNewEvent(mc_i)$
11                 $t \leftarrow tm.getEventFromTailOf(mc_1..mc_{i-1})$
12                 if $t$ is not defined then $t \leftarrow [$;
13                 $tm.addEdge(t, e)$
14             end
15         end
16         $e \leftarrow tm.getEvent(mc_n)$
17         $tm.addEdge(e, ])$
18     end
19 end
```

The following example shows the main cases treated by the algorithm; each line represents a test case and the four test cases are in the same test class. Only method `c()` of PageObject `po2` has parameters.

tc_1 – `po1.a(),po1.b(),po2.c(param1: value1)`

tc_2 – `po1.a(),po1.b(),po2.c(param1: value2)`

tc_3 – `po1.a(),po1.b(),po2.d(),po2.e()`

tc_4 – `po2.f(),po2.g()`

Figure 3.4 shows how these four test cases are mapped in an ESG model. Test cases tc_1 and tc_2 are represented as the same event sequence, being distinguished by different parameter values in `po2.c()`. Test case tc_3 has a common prefix with tc_1 and tc_2 (namely, `po1.a(),po1.b()`), so the remaining suffix is appended to the event that represents the tail of the common prefix. Test case tc_4 has no common prefix, so a different branch is added to initial node [.

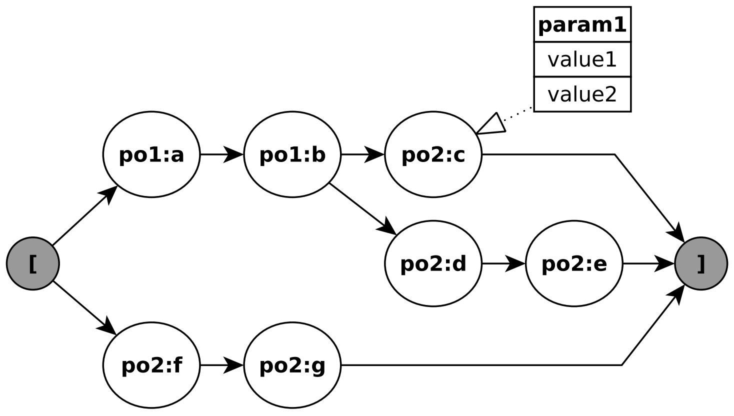


Figure 3.4: Example of a test model.

Figure 3.5 illustrates the ESG inferred from the test case in Figure 3.3. Special labels are given to some events: constructor call for the `HomePage` class has an event with the prefix “OPEN”, and assertions (like `assertEquals`) has a different representation.

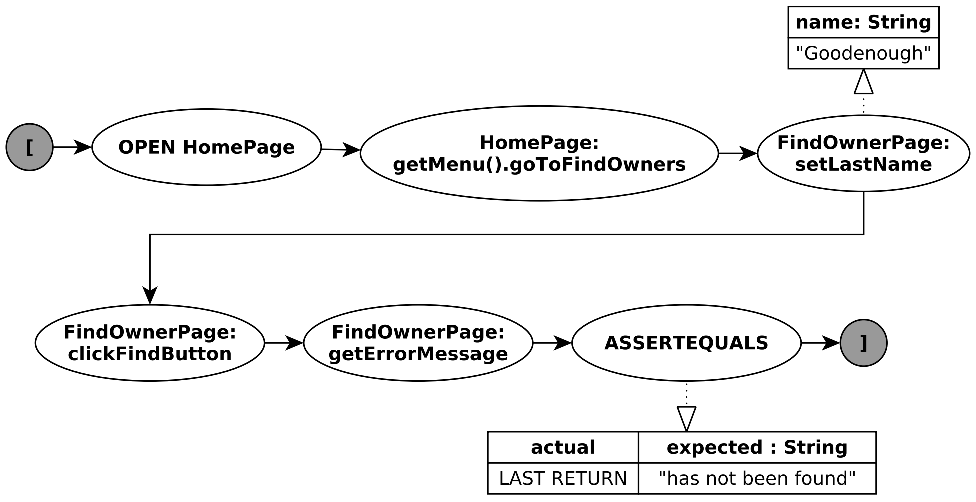

name: String
"Goodenough"
[
OPEN HomePage
HomePage: getMenu().goToFindOwners
FindOwnerPage: setLastName
FindOwnerPage: clickFindButton
FindOwnerPage: getErrorMessage
ASSERTEQUALS
]
actual
expected : String
LAST RETURN
"has not been found"

Figure 3.5: Test model inferred from existing test case.

3.3 MODEL EXTENSION

In this step, the tester can extend the test models inferred previously. Moreover, new test models can be designed for different purposes. Nevertheless, Rules (1), (2) and (3) are enforced during modeling time.

To assure Rule (1), model elements inferred from existing tests (previous step) cannot be modified by the tester at model level. If needed, such changes must be performed by developers at code level. Rules (2) and (3) treat the modifications in the model that involve the insertion of events not mapped by any PageObject; this reflects on changes at code level. In summary, there are two types of changes.

The first type happens when *the tester adds an event that is not mapped to the PageObjects' API*. Figure 3.6 exemplifies it with an event and two edges in red. The tester adds the red event, but class `FindOwnerPage` has no method `clearTextFields()`. In this case, the change at model level also provokes a change at code level, and an abstract test case is generated.

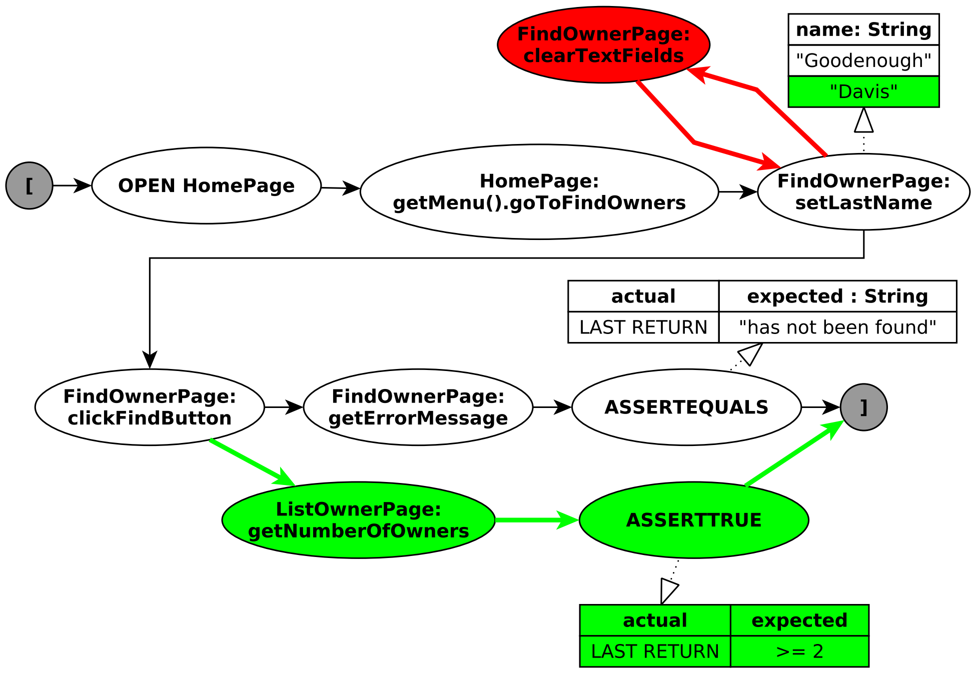

FindOwnerPage: clearTextFields
name: String
"Goodenough"
"Davis"
[
OPEN HomePage
HomePage: getMenu().goToFindOwners
FindOwnerPage: setLastName
actual
expected : String
LAST RETURN
"has not been found"
FindOwnerPage: clickFindButton
FindOwnerPage: getErrorMessage
ASSERTEQUALS
]
ListOwnerPage: getNumberOfOwners
ASSERTTRUE
actual
expected
LAST RETURN
>= 2

Figure 3.6: Example of possible changes.

The second type of change happens when *the tester modifies the model but do not alter the PageObjects' API*. For instance, she/he adds edges, insert events already implemented by the PageObjects, or include more test data (that we call event instances). Figure 3.6 exemplifies this scenario with events and edges in green. Such changes trigger the generation of test cases that are directly executed in the SUT. Further details in Section 3.4.

3.4 TEST GENERATION

This step focuses on generating abstract and concrete test cases. Abstract test cases are failing test cases that cover events not mapped by the PageObjects' API. As the PageObjects' API needs to be modified and the development team needs to have an example, automated test cases are generated to guide the development team in implementing the PageObject classes.

Figure 3.7 shows the abstract test case for the changes in red in Figure 3.6. Lines 1-11 presents the test case introduced in the code base and the modification in class `FindOwnerPage` in Lines 15-17. This is an executable but failing test case that will generate a demand for the development team to implement.

```
1  @Test
2  public void testEventClearTextFields() {
3    HomePage homePage = new HomePage(driver);
4    FindOwnerPage findOwnerPage = homePage.getMenu()
5      .goToFindOwners();
6    findOwnerPage.setLastName("Goodenough");
7    findOwnerPage.clearTextFields();
8    findOwnerPage.setLastName("Goodenough");
9    findOwnerPage.clickFindButton();
10   assertEquals("has not been found", findOwnerPage.getErrorMessage());
11 }
12
13 Class FindOwnerPage {
14   ..
15   public void clearTextFields() {
16     throw new NotImplementedException();
17   }
18 }
```

Figure 3.7: Abstract test case to be concretized.

In this step we also generate concrete test cases. We call concrete test cases, the generated test codes where their respective PageObjects methods are already implemented. Therefore, once this type of test case is generated from the model, it is ready to be run against the SUT.

Algorithm 2 describes a procedure to generate test cases (method calls sequences) covering the new model elements (vertices, edges and event instances) created by the tester. Event instances contains the parameters related to an event (node method). The algorithm receives as input a valid ESG test model and the output are valid test cases. The algorithm is divided in two parts. The first part (lines 2-13) generates new test cases to cover the new event instances created by the tester. In line 3, all the distinct group names for event instances are queried. We define a group name for event instances (parameters) that, after test cases generation, must belong to the same test case. In line 4, the algorithm iterates over these groups. Line 5 queries all event instances to define which vertices must be included in the new test case. Line 6 gets the must pass vertices configured in the model, as restrictions, by the tester. Line 9 defines a valid sequence that includes the vertices queried in lines 5-6. Then, lines 10-12 create the new method and applies the event instances (parameters) to the corresponding methods. In this way, the algorithm generated a test case covering all the vertices that contains event instances of the group. The second part (lines 14-26) covers the new edges, hence the algorithm will also cover the new vertices inserted by the tester. In line 14 the algorithm retrieves all new edges inserted by the tester, and not covered in the first part of algorithm (before line 13), then run through the list generating CES that cover the respective edge, until all edges are covered. In line 16, the first uncovered edge of the *notCoveredEdges* array is selected and its origin/target vertices are queried in lines 17-18. Line 19 defines the must pass vertices configured in the model by the tester. Line 21 finds a valid sequence containing the origin/target vertices of the edge and the must pass vertices. Lines 22-23 extract the methods of the sequence and define the event instances that will be injected in those that require parameters. In line 25 the *notCoveredEdges* array is updated. The algorithm implemented to find the path that covers all necessary vertices is based in the Breadth-First Search algorithm (CORMEN et al., 2009).

3.5 IMPLEMENTATION

To check the feasibility of the proposed approach, we implemented a tool, called MoLeWe (*MOdel-based testing LEveraged for WEb tests*) that supports the three steps described. To make it actionable, MoLeWe was initially designed for web application test suites written in Java, using Selenium WebDriver along with the PageObjects pattern. The MoLeWe tool was developed on top of the FourMa[1] tool, which supports the MBT of mobile Android apps (FARTO; ENDO, 2017).

In the model inference step, we used the Spoon (PAWLAK et al., 2016) parser to process Java test files and build the model. For each method or constructor call in Java test cases, the tool creates a new node (event) in the ESG model. The method call's arguments are mapped into event instances, which allows the reuse of node with other parameters (i.e., event instances). As we aimed to generate a simple and readable model, MoLeWe only considers method calls and constructors to produce model elements, avoiding lines with variable assignments and other statements that do not improve the test suite's understanding through the model. This resulted in smaller models, concentrated with information that are useful to understand the SUT.

[1] https://github.com/andreendo/FourMA

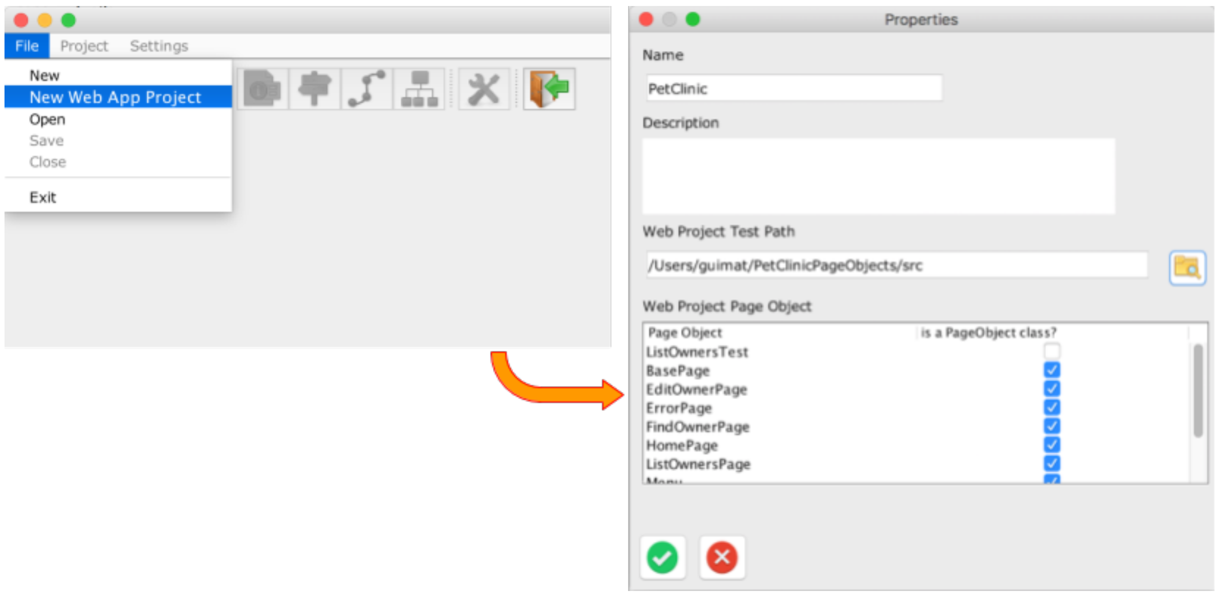
Algorithm 2: Generating test cases

```
input  : Valid ESG test model $TM$
output: Test cases $TCases$
1  $TCases \leftarrow \{\}$
2  $eventInstances \leftarrow TM.getNewEventInstances()$
3  $eventInstancesGroups \leftarrow getDistinctTCGroupName(eventInstances)$
4  foreach $eig \in eventInstancesGroups$ do
       /* get event instances in the same TC                  */
5      $eivs \leftarrow TM.getEventInstancesVertexFromGroup(eig)$
6      $othersMustPassVertices \leftarrow TM.getRestrictions(eig)$
7      $eivs.append(othersMustPassVertices)$
8      $verticesSequence = []$
       /* get order using BFS based algorithm                */
9      $verticesSequence = TM.getOrderBFS(eivs)$
10     $newTCPath \leftarrow TM.getPathFromSeqArray(verticesSequence)$
11     $solveEventInstancesForPath(newTCPath)$
12     $TCases.add(newTCPath)$
13 end
14 $notCoveredEdges \leftarrow TM.getNewEdges()$
15 while not empty(notCoveredEdges) do
16     $e \leftarrow notCoveredEdges[0]$
17     $a \leftarrow TM.getEdgeOriginVertex(e)$
18     $b \leftarrow TM.getEdgeTargetVertex(e)$
19     $mustPassVertices \leftarrow TM.getRestrictions(e)$
20     $verticesSequence = []$
       /* get order using BFS based algorithm                */
21     $verticesSequence = TM.getOrderBFS(mustPassVertices, a, b)$
22     $newTCPath \leftarrow TM.getPathFromSeqArray(verticesSequence)$
23     $solveEventInstancesForPath(newTCPath)$
24     $TCases.add(newTCPath)$
25     $updateNotCoveredEdges(notCoveredEdges)$
26 end
```

In the model extension step, MoLeWe allows the manipulation and augmentation of the model inferred in the previous step, inserting new nodes, edges or input data as event instances. Moreover, the tool supports the configuration of different kinds of restrictions to the test cases' paths, like selecting the nodes a test case must include.

In the test generation step, when executing Algorithm 2 (see Section 3.4) along with the Spoon parser support, new abstract and concrete test cases are generated according to model extensions performed by the tester. As output, MoLeWe generates the Java test classes containing the new test cases. The tests that involve newly-introduced events will fail and have to be concretized to be successfully executed in the SUT.

To illustrate how the tool works, the PetClinic application is used as an example. When opening the tool, the first step is to create a new project. Figure 3.8 illustrates how to access the new project screen containing the project properties and how the form must be filled in for the running example. When filling in the fields and informing the path to the test project to be used, MoLeWe identifies the classes present in the project and suggests possible PageObjects classes. The user must confirm the classes that really are PageObjects and click on the green button to create a new project.



Name

PetClinic

Description

Web Project Test Path

/Users/guimat/PetClinicPageObjects/src

Web Project Page Object

Page Object	is a PageObject class?
ListOwnersTest	☐
BasePage	✓
EditOwnerPage	✓
ErrorPage	✓
FindOwnerPage	✓
HomePage	✓
ListOwnersPage	✓

Figure 3.8: Creating a new project

The tool then infers the model for the informed testing project, as shown in Figure 3.9. The first test case to be created will be the search for a nonexistent user. Figure 3.10 shows this procedure. First, we create a node with the corresponding assert statement (*assertTrue*). A double click on the screen opens a dialog to enter a description for the new node. Then, the new node is created, but still without the relation with the corresponding PageObject method (blue node). To relate the node to the corresponding PageObject method, a right-click on the screen shows the methods available in PageObjects and some assertion options (*assertEquals* and *assertTrue*). When selecting *assertTrue*, the node turns yellow, indicating that the method is now related.

The next step is to insert the parameter (event instance) in the statement *setLastName*. Figure 3.11 shows this procedure. A right-click on the desired node shows the option to access the parameter editing screen of the selected node method. The *Add row* button at the bottom of the screen adds a new empty parameter row (event instance) to the table. The first column is the event instance ID, which should not be changed by the user. The second column shows the *lastName* parameter, which must be changed with the desired value, in this case it is “Davis”. In

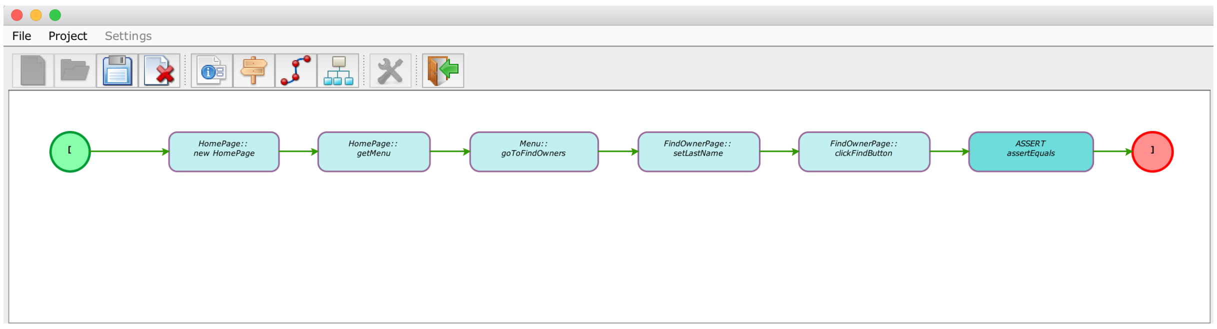


Figure 3.9: Inferred model

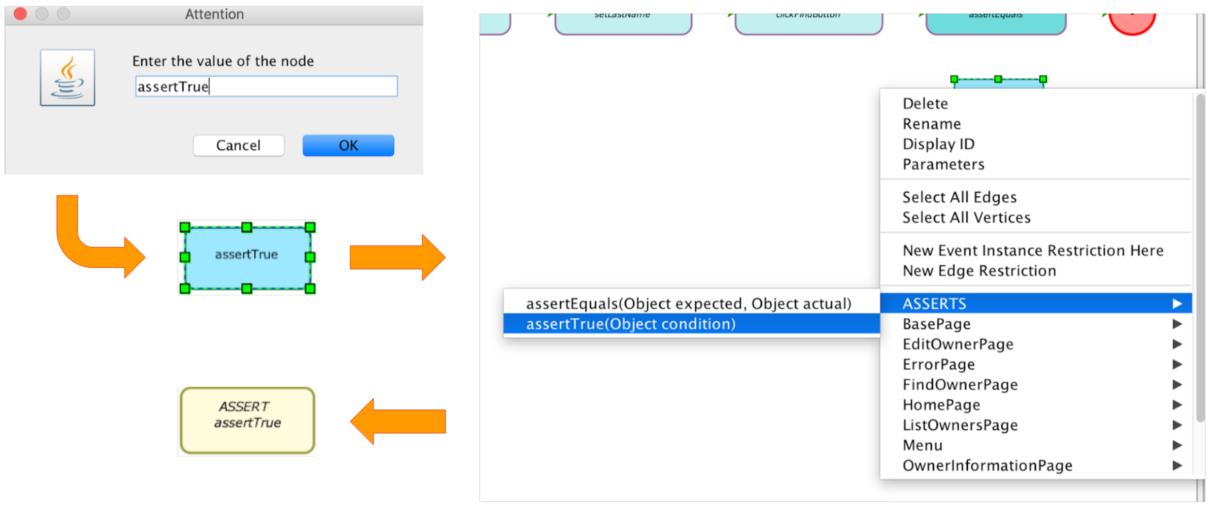
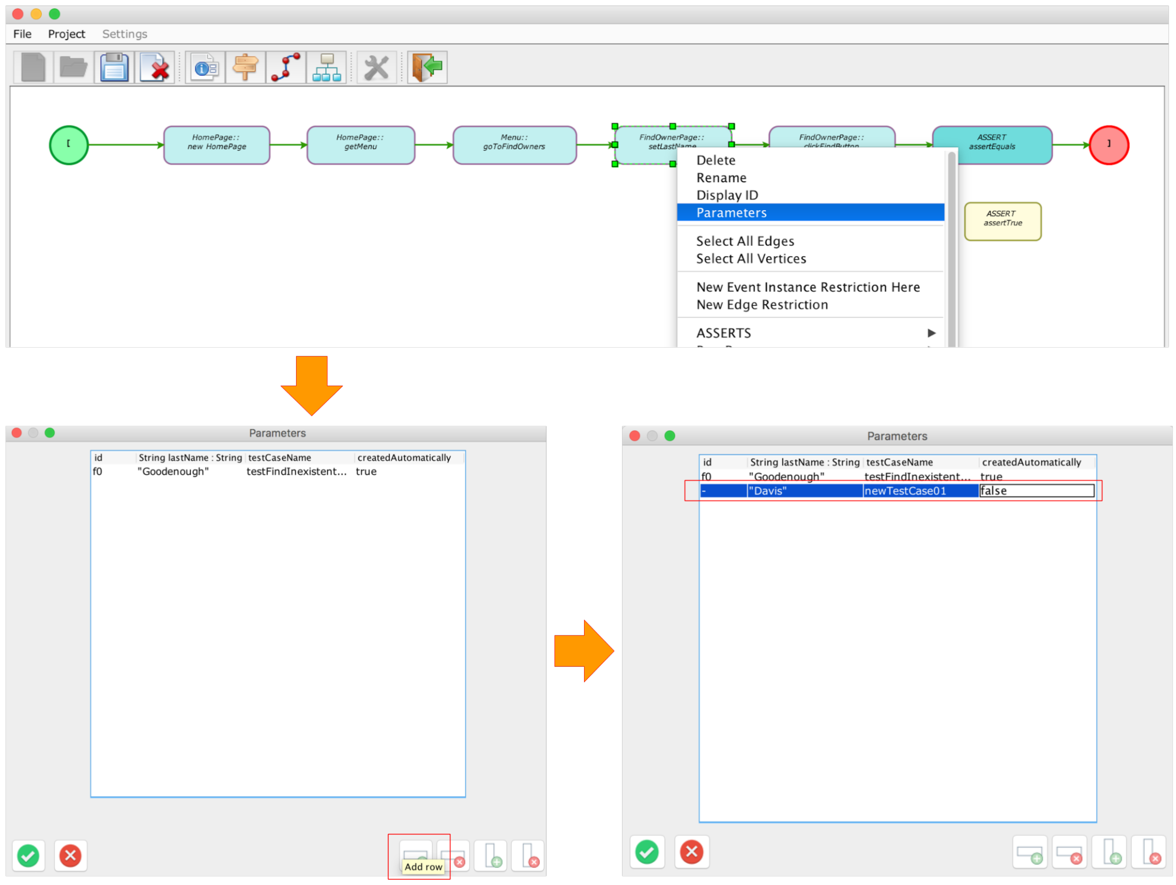


Figure 3.10: Create assert node

the third column, a name must be inserted for this event instance, which will serve to guide the tool, allowing the generation of a single test case that contains all nodes that have an event instance with the same name. The fourth column reports that this event instance was not generated automatically, but by the user.

After setting the parameter for the *setLastName* method, it is also necessary to configure the parameter of the *assertTrue* statement, repeating the same procedure.

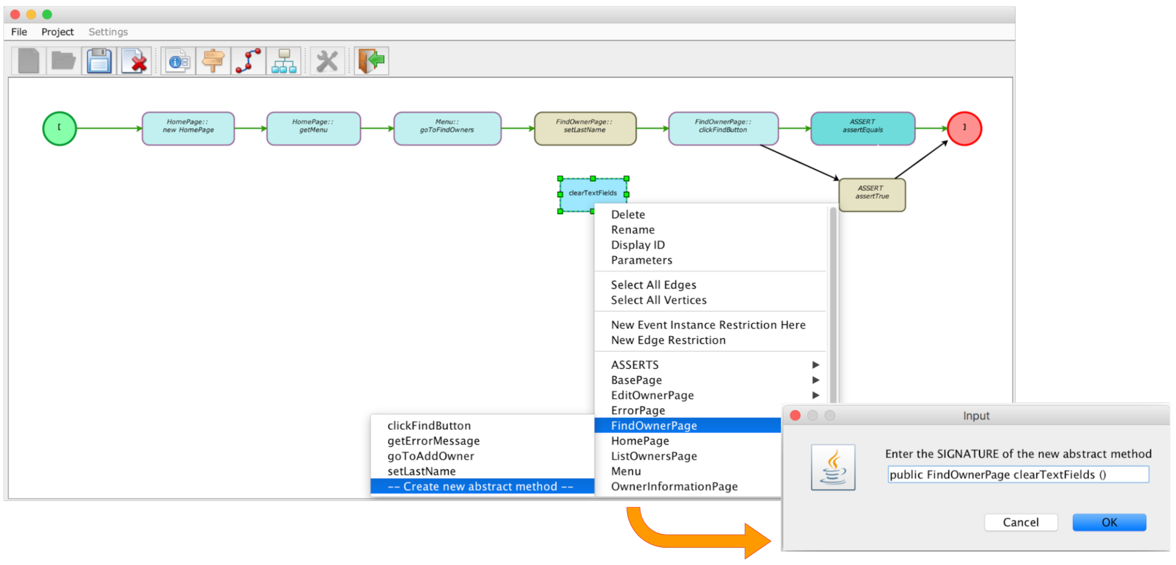
With the nodes created and the parameters configured, it remains to create the edges that connect the new nodes and form the desired test case. The first test case was created, now we are going to create the node with the *clearTextFields* method for the second test case. In this case, *clearTextFields* does not exist in the PageObjects classes. Therefore, it is necessary to create a new abstract method. Figure 3.12 illustrates the procedure for creating a new abstract method in the PageObject class called *FindOwnerPage*. To create the abstract method, it is necessary to insert the method signature, including the return type and the function parameters. After creating the abstract method, relate it to the created node, as shown in Figure 3.13. The tester can also create new PageObjects classes, for web pages that do not have a PageObject



id	String lastName : String	testCaseName	createdAutomatically
f0	"Goodenough"	testFindInexistent...	true
-	"Davis"	newTestCase01	false

Figure 3.11: Clicking in edit parameters (event instances) button

class created in the test suite.



Enter the SIGNATURE of the new abstract method

public FindOwnerPage clearTextFields ()

Figure 3.12: Creating new abstract method

Finally, insert the edges connecting the new node to form the new test case. With the

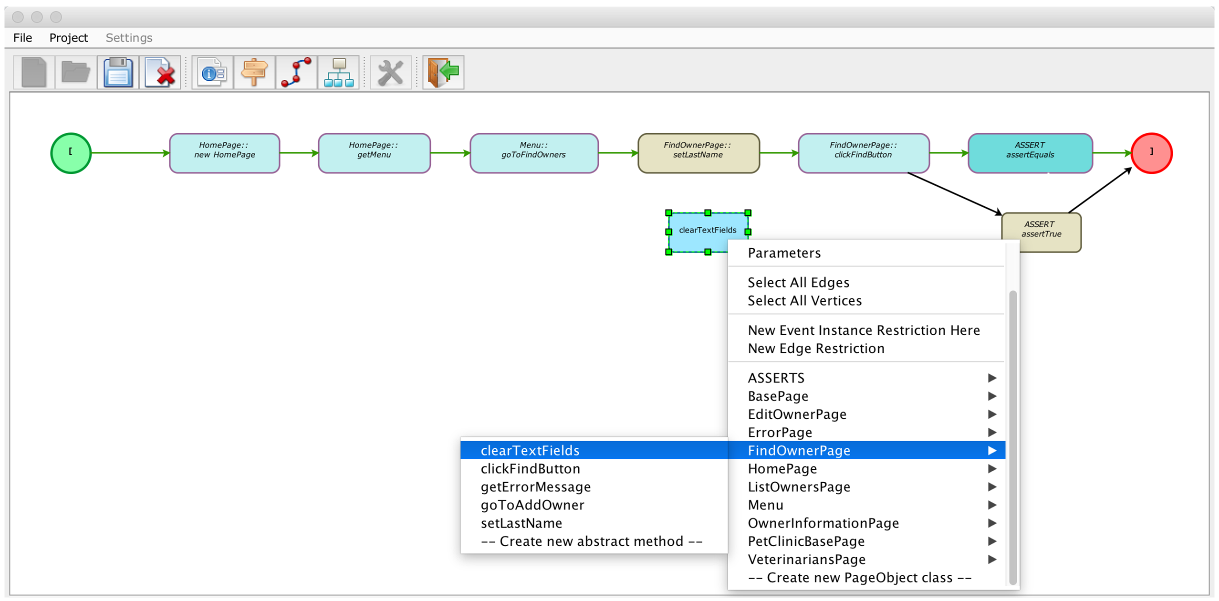
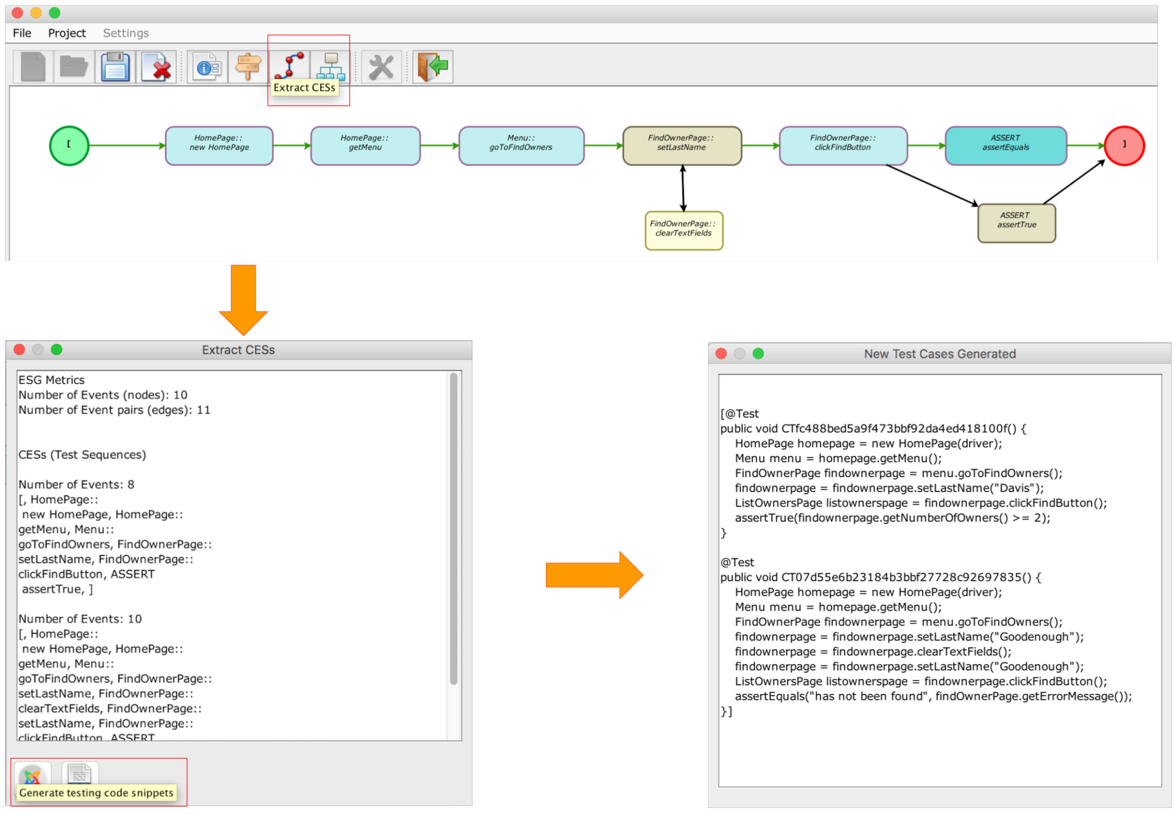


Figure 3.13: Relating the node with the new abstract method

test cases modeled, Figure 3.14 shows how to generate the new test cases. On the *Extract CES* screen, clicking on *Generate testing code snippets* the new modeled test cases script is displayed on a new screen. This code must be moved and integrated into the existing test suite.



```
ESG Metrics
Number of Events (nodes): 10
Number of Event pairs (edges): 11


CESs (Test Sequences)

Number of Events: 8
[, HomePage::
 new HomePage, HomePage::
getMenu, Menu::
goToFindOwners, FindOwnerPage::
setLastName, FindOwnerPage::
clickFindButton, ASSERT
 assertTrue, ]

Number of Events: 10
[, HomePage::
 new HomePage, HomePage::
getMenu, Menu::
goToFindOwners, FindOwnerPage::
setLastName, FindOwnerPage::
clearTextFields, FindOwnerPage::
setLastName, FindOwnerPage::
clickFindButton, ASSERT
```

```
[@Test
public void CTfc488bed5a9f473bbf92da4ed418100f() {
   HomePage homepage = new HomePage(driver);
   Menu menu = homepage.getMenu();
   FindOwnerPage findownerpage = menu.goToFindOwners();
   findownerpage = findownerpage.setLastName("Davis");
   ListOwnersPage listownerspage = findownerpage.clickFindButton();
   assertTrue(findownerpage.getNumberOfOwners() >= 2);
}

@Test
public void CT07d55e6b23184b3bbf27728c92697835() {
   HomePage homepage = new HomePage(driver);
   Menu menu = homepage.getMenu();
   FindOwnerPage findownerpage = menu.goToFindOwners();
   findownerpage = findownerpage.setLastName("Goodenough");
   findownerpage = findownerpage.clearTextFields();
   findownerpage = findownerpage.setLastName("Goodenough");
   ListOwnersPage listownerspage = findownerpage.clickFindButton();
   assertEquals("has not been found", findOwnerPage.getErrorMessage());
}]
```

Figure 3.14: Generating the new test cases

This prototype tool was implemented in order to validate the approach. Therefore, the tool is not prepared with all the existing statements in Java that are used in the creation of test cases with Selenium WebDriver. For example, only two types of assertions (*assertTrue* and *assertEquals*) were included in the tool, which were sufficient to reproduce the experimental study. Models can also only be extended with invocations to methods of PageObjects classes, so that it is not possible to insert, as a node in the model, assignments to variables, conditional or repetition structures, for example. This structure took place in order to simplify the manipulation and readability of the generated model.

The algorithm used to generate the new test cases is not optimized, since it is based on the Breadth-First Search algorithm, with the criterion of covering all edges. If the tester inserted cycles into the model, the algorithm will cover the loops by going through them only once. In order to allow flexibility in the modeling of test cases, the possibility of modeling some restrictions has been implemented. The tester can insert two types of restrictions in the model to customize the generation of test cases. In the first, when inserting a new edge in the model, the tester can choose vertices that should be part of the test case that will be generated to cover the edge. In the second, he can select vertices that should be part of the generated test case to cover an event instance group. These two restrictions were sufficient for the extensions that were modeled during the experiment.

The source code of the MoLeWe tool and the experimental artifacts, like the raw data collected, the participants' test projects, inferred models, the virtual machines with the web applications, and experimental instructions are available at: `https://github.com/guimattiello/MoLeWe`

3.6 FINAL REMARKS

This chapter presented the approach proposed that uses existing test suites to facilitate the adoption of MBT and the tool (MoLeWe) implemented to validate it. The approach consists of three steps: inferring the model, extending the model and generating new test cases. All the steps involved were explained and exemplified. We introduced MoLeWe with an usage example, reproducing the approach with the PetClinic's find owners test case. Due to time constraints, the tool currently supports only test projects developed with the Java language and within certain code standards, as mentioned in this chapter. The next chapter presents the evaluation of the approach and the results obtained when applying the tool.

4 EVALUATION

This chapter presents the evaluation of the proposed approach. Section 4.1 introduces the study setting, showing how the evaluation was carried out and which research questions it sought to answer. Section 4.2 presents the results obtained, divided by the approach' steps. Section 4.3 raises a discussion of the results and Section 4.4 lists the threats to validity.

4.1 STUDY SETTING

To evaluate the feasibility and usefulness, we propose seven research questions divided by the three main steps of the proposed approach.

For the model inference step, we set out the following research questions (RQs):

- **RQ**$_{1.1}$ Can the approach infer models from real-world test suites?
- **RQ**$_{1.2}$ What is the performance cost of this step?

For RQ$_{1.1}$, we intend to verify if understandable models can be automatically produced from test suites; while RQ$_{1.2}$ aims to analyze the performance in CPU time and the scalability of this step.

For the model extension step, the following questions were defined:

- **RQ**$_{2.1}$ Can the inferred models be extended to test different scenarios?
- **RQ**$_{2.2}$ What is the effort involved in the model extensions?

RQ$_{2.1}$ focuses on the adequacy of inferred models to be extended with untested scenarios. As for RQ$_{2.2}$, we intend to assess the human effort to include such extensions into the test model.

Concerning the test generation step, we proposed the following questions:

- **RQ**$_{3.1}$ What is the performance cost of generating new tests from the extended model?
- **RQ**$_{3.2}$ What is the effort to concretize the abstract test cases?
- **RQ**$_{3.3}$ How much do the new test cases impact on code coverage?

$RQ_{3.1}$ aims to analyze the performance in CPU time and the scalability of generating new tests. For $RQ_{3.2}$, we analyze the human effort to make the abstract test cases executable in the SUT. Finally, $RQ_{3.3}$ intends to shed some light on the new test cases' code coverage.

For this study, nine open source web applications were selected. We prioritized web applications that have a GitHub repository, an active community (observed by recent commits, forks, and number of stars), varied sizes (ranging from 9.592 to 1,784.595 KLoC), and different domains. We opted by web applications whose server side code are written in the same programming language, namely PHP; this helps the comparison with respect to code coverage. Table 4.1 summarizes the selected applications; for each one, it shows its ID, name, KLoC, number of stars, commits, forks, GitHub link, and a brief description.

ID	Name	KLoC	#Stars	#Commits	#Forks	Github project link	Description
1	Akaunting	98.885	1619	1468	593	https://github.com/akaunting/akaunting	Accounting software for small businesses.
2	Attendize	379.546	2421	876	608	https://github.com/Attendize/Attendize	Ticket selling and event management platform.
3	sysPass	200.551	490	1530	103	https://github.com/nuxsmin/sysPass	Multi-user password manager for business and personal use.
4	Firefly	263.756	2027	11541	342	https://github.com/firefly-iii/firefly-iii	Personal finances manager.
5	Laravel-Gymie	129.287	169	55	99	https://github.com/lubusIN/laravel-gymie	Member management system for gyms and clubs.
6	Lychee	9.592	4788	1594	625	https://github.com/electerious/Lychee	Easy-to-use photo-management system.
7	Mapos	251.270	201	242	199	https://github.com/RamonSilva20/mapos	Controlling system for work orders.
8	MediaWiki	1,784.595	1442	87349	686	https://github.com/wikimedia/mediawiki	Wiki engine.
9	OpenCart	614.254	4671	9285	3653	https://github.com/opencart/opencart	Shopping cart and online store management system.

Table 4.1: Sample web applications

We evaluated the approach and its supporting tool with 18 undergraduate students of an advanced course on software testing; they previously took and introductory course on software testing. The participants received 10 hours of training, guidelines and practice on using the Selenium WebDriver framework and the PageObjects pattern to develop Java-based Web tests. Each application in Table 4.1 was randomly assigned to two participants. Each participant was asked to create a test suite with at least ten test cases for the main features of the application. Some refactorings were then performed to attend the guidelines and make the implementations anonymous. We used these 18 test suites to evaluate the three steps of the approach, implemented by MoLeWe. Once we had the 18 test suites, the execution of the three steps of the approach was conducted by the authors of this work, as described below.

From this diverse set of test suites, MoLeWe inferred 18 ESG models. The models were extended and augmented (through the tool) with the insertion of new nodes, edges and event instances, seeking to generate new test cases. To devise potential extensions in the model, we analyzed, per application, the two test suites developed independently by the participants X and Y. The idea was to extend the participant X's model to subsume participant Y's tests; this step was repeated now starting with participant Y's model. This procedure was performed for the 18 test suites and, whenever possible, reusing existing elements and methods.

Using the extended model, MoLeWe generated new abstract and concrete test cases, as specified in Algorithm 2. Then, the generated PageObjects' methods, derived from abstract events in the model, were concretized. After this concretization step, the test suite was run against the SUT. As we analyzed the back-end execution and we are dealing with mature applications that do not have many bugs, line coverage was chosen as a proxy to investigate the effectiveness of the proposed approach. We collected metrics like line coverage, projects LoC and number of new test cases. We used the JetBrains plugin MetricsReloaded[1] to collect project metrics (e.g., LoC), and we configured the web applications to collect line coverage of the test suites[2]. We instrumented MoLeWe to collect the CPU time in the model inference and test generation steps. We also measured the elapsed time of manual tasks related to model extension and test concretization. The experiments were performed on a 2.7 GHz Intel Core i5 and 8 GB RAM with macOS High Sierra.

4.2 ANALYSIS OF RESULTS

In this section, we analyze the obtained results in the conducted experimental evaluation.

4.2.1 INFERENCE

For $RQ_{1.1}$, Table 4.2 summarizes the results of the inferred models, for each of the 18 test suites developed by the participants. For each project, the table shows the number of test cases, LoC of the entire project, LoC of the test classes, number of PageObject classes, and number of PageObjects methods, as well as metrics related to the inferred model (namely, number of nodes, edges, and event instances). By manually inspecting the test suites and their inferred models, we observed that MoLeWe correctly extracted the test models. This brings initial evidence that the approach can be used to infer test models from real-world web tests.

We noticed that the adoption of PageObjects pattern makes it more readable and facilitates its manipulation, since specific details of the Selenium WebDriver API are abstracted by the PageObject classes. The adequacy of the generated model was also verified in the model extension step, since in most cases just reading the inferred model X was enough to understand

[1]https://plugins.jetbrains.com/plugin/93-metricsreloaded
[2]https://github.com/tarunlalwani/php-code-coverage-web

ID	Participant	Projects Metrics					Inferred Model		
		#Test Cases	Project LoC	Tests LoC	#PO Classes	#PO Methods	#Nodes	#Edges	#Event Instances
1	A	16	1483	657	15	112	101	115	120
	B	10	1565	517	18	151	70	78	51
2	C	10	1217	520	12	81	89	97	131
	D	10	687	97	14	73	53	61	30
3	E	10	777	160	10	84	72	80	47
	F	12	1451	537	17	135	97	108	58
4	G	10	1125	440	13	90	59	67	49
	H	13	1186	314	24	115	141	152	137
5	I	10	1237	477	16	82	66	74	72
	J	10	821	86	21	71	46	54	40
6	K	10	563	112	5	62	87	94	39
	L	10	525	154	7	33	39	47	59
7	M	10	844	325	9	70	65	73	68
	N	11	1228	377	7	75	134	143	87
8	O	10	699	170	15	68	55	63	53
	P	10	484	138	11	43	46	54	42
9	Q	10	1378	221	43	124	117	125	67
	R	10	1213	153	34	96	78	86	48

Table 4.2: Data About Model Inference

and extract the complementary test scenarios and apply them to the inferred model Y. The exceptions were when the participant created methods in the PageObject classes that performed many actions, making it difficult to understand the purpose and making it necessary to look at the source code to understand it.

Answering $RQ_{1.2}$, the runtime[3] required to infer the model of the smallest project with 484 LoC is on average 697 milliseconds and the longest runtime of 2309 milliseconds was related to a project with 1378 LoC. According to Figure 4.1, the runtime to infer the model has some relation to the project LoC and, observing the linear regression ($y = 419.466 + 0.661x$) illustrated by the red dashed line, seems to grow linearly[4]. This result is expected, since the model's inference algorithm runs through the methods of the test classes sequentially and does not have non-linear characteristics. This provides some evidence that the approach' step of model inference is scalable. Each point on the graph represents a project.

4.2.2 MODEL EXTENSION

To answer $RQ_{2.1}$, Table 4.3 summarizes the model metrics (number of test cases, nodes, edges and event instances) after the model extension step. On average, the number of

[3]We collected the average runtime in three runs for each model.
[4]We ordered the test projects by LoC to plot the graph.

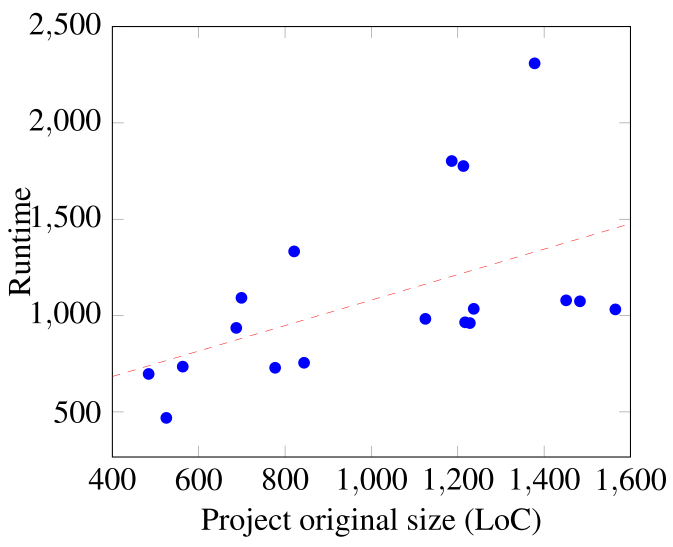


Figure 4.1: Runtime to infer the model.

edges increased by 61.39%, nodes by 56.24%, and event instances by 70.63%. Some projects of the same application had a big difference in the addition of the new elements. In application #6, project K had 16.09% of nodes added, 22.34% of edges added and 46.15% of events instances added. On the other hand, project L had 115.38% of nodes added, 112.77% of edges added and 89.83% of events instances added. This happened because the projects of the same application have test cases with different styles. While one project focused on test cases with long sequence of events, the other prioritized shorter test cases; this did not necessarily reflect in the line coverage. Thus, when the complementary test cases of project K (long test cases) were modeled in project L (short test cases), a greater number of new elements in the model were needed to reproduce the same test. We observed that all extensions extracted from the pair of projects could be modeled, making it possible to produce new tests that exercise scenarios that were not previously tested. The delta (Δ) represents the difference between the number of elements (nodes, edges and event instances) before and after the model extension step.

The required effort to extend the model ($RQ_{2.2}$) can be measured in number of edges, nodes, and event instances added to the model, as well as the time elapsed on this task. To do so, we assume that the tester already knew which test cases, how they should be modeled, and the structure of the project being extended. The last column of Table 4.3 presents the time spent to extend the models, per project. The time varied from 15 to 88 minutes, depending on the number of new elements that needed to be modeled. We observed that the modeling effort was reasonable, the models were easily extended, and further benefits can be obtained from the new test cases that are generated (see the analyses for following RQs).

ID	Participant	Model AFTER Extension				Results						
		#Test Cases	#Nodes	#Edges	#Event Instances	Δ Nodes	% of Nodes Added	Δ Edges	% of Edges Added	Δ Event Instances	% of Event Instances Added	Effort Time (min.)
1	A	22	111	129	148	10	9.90%	14	12.17%	28	23.33%	15
	B	23	102	116	103	32	45.71%	38	48.72%	52	101.96%	25
2	C	18	125	145	188	36	40.45%	48	49.48%	57	43.51%	29
	D	18	84	100	56	31	58.49%	39	63.93%	26	86.67%	22
3	E	22	126	148	91	54	75.00%	68	85.00%	44	93.62%	50
	F	21	122	144	87	25	25.77%	36	33.33%	29	50.00%	35
4	G	21	115	131	113	56	94.92%	64	95.52%	64	130.61%	50
	H	21	151	171	176	10	7.09%	19	12.50%	39	28.47%	19
5	I	19	125	145	126	59	89.39%	71	95.95%	54	75.00%	48
	J	19	73	90	67	27	58.70%	36	66.67%	27	67.50%	34
6	K	16	101	115	57	14	16.09%	21	22.34%	18	46.15%	17
	L	18	84	100	112	45	115.38%	53	112.77%	53	89.83%	44
7	M	19	100	120	109	35	53.85%	47	64.38%	41	60.29%	44
	N	21	163	184	127	29	21.64%	41	28.67%	40	45.98%	43
8	O	19	77	93	105	22	40.00%	30	47.62%	52	98.11%	38
	P	18	73	88	70	27	58.70%	34	62.96%	28	66.67%	28
9	Q	20	198	216	111	81	69.23%	91	72.80%	44	65.67%	68
	R	19	181	198	95	103	132.05%	112	130.23%	47	97.92%	88
Average:						**38.67**	**56.24%**	**47.89**	**61.39%**	**41.28**	**70.63%**	
Minimum:						**10**	**7.09%**	**14**	**12.17%**	**18**	**23.33%**	
Maximum:						**103**	**132.05%**	**112**	**130.23%**	**64**	**130.61%**	

Table 4.3: Data About Model Extension

4.2.3 TEST GENERATION

Regarding the runtime to generate the new test cases, raised by $RQ_{3.1}$, the average time required to extract the paths from the ESG and produce the new test cases is 343 milliseconds, ranging from 90 to 575 milliseconds. Figure 4.2 illustrates the runtime[5] to generate new test cases as a function of the number of new elements added[6] in the model extension step. Each point on the graph represents a project. Notice that, according to the linear regression ($y = 161.885 + 1.416x$) represented by the red dashed line, the growth gives evidence that this step scales to even greater modifications in the model or larger projects and test suites, since the test case generation algorithm is based on Breadth-First algorithm, which runs in time linear in the size of the adjacency-list (CORMEN et al., 2009).

Table 4.4 shows that most of the test cases' events generated by the model extension step are from concrete methods, prevailing the reuse of the PageObject methods. On average, 70.54% of the new events are concrete events, fact that considerably reduces the effort in generating new test cases. For instance, project B of application #1 needed only one event to be concretized (0.58%); it means that the test suite generated by MoLeWe was almost ready to be executed against the SUT. In fact, the participant implemented unused PageObjects methods, which made it possible to take advantage of them in the model extension. Project P of application #8 had the lowest percentage of concrete events in relation to new events (49.02%). This means that the tester needs to implement just over half of the new events.

[5]The runtime is an average of three runs for each model.

[6]We sum the number of events, edges, and event instances.

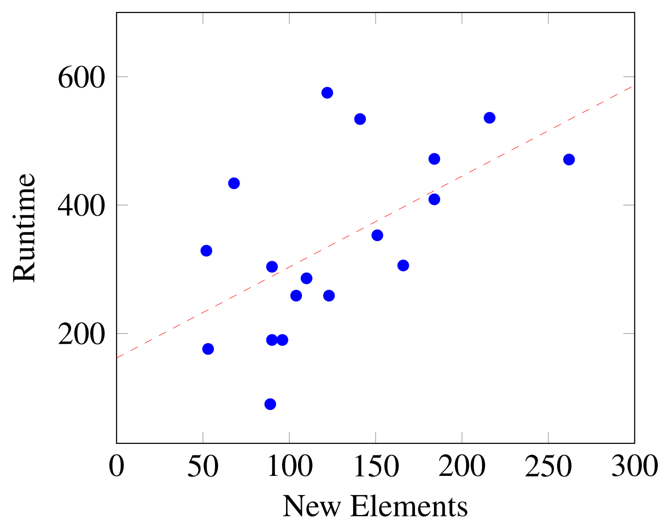


Figure 4.2: Runtime to generate new test cases.

To answer $RQ_{3.2}$, the effort time (in minutes) to concretize the abstract methods is shown in Table 4.4. This time is proportional to the number of abstract methods that need to be implemented, so projects with more abstract methods (projects E, G, I, Q, R) took more time to be concretized. In addition, this represents an effort that is not unique to the approach, as it is part of the implementation of any test suite.

We observed that the gain in line coverage depends on how the test suites are structured, the amount of PageObject methods implemented, and the SUT properties. In general, as shown in Table 4.4, the line coverage of the original test suite produced by the participants is low. This occurs due to the varying size of the projects and the limited number of test cases per participant. For example, in smaller applications like application #6, higher line coverage was achieved (up to 41.24%). On the other hand, in larger applications like applications #2 and #9, the test suites did not achieve such a high coverage (around 6%). As expected, the coverage achieved by the two extended models of the same application was similar (see 8th column of Table 4.4). Small differences were observed due to the input data used and the reuse of existing functions.

The last column of Table 4.4 shows the line coverage gains of the test suite generated from the extended model. To answer $RQ_{3.3}$, we can highlight some cases in which line coverage had a large increase, such as project N of application #7 (393.94%). This occurred due to the fact that the participant created test cases that did not cover large chunks of code, resulting in a small initial line coverage (1.98%). In addition, Project M encompassed many test cases that were not developed in Project N, achieving greater coverage. Thus, by bringing the complementary test cases from project M to project N, project N had a huge increase in line coverage, jumping from 1.98% to 9.78%. On the other hand, project K of application #6 obtained a small increase (only 4.00%). This happened because almost all of the test cases in project K have already been covered by the test cases in project L, which has not left many test cases in project L to increase the coverage of project K. On average, the increase in line coverage was 38.97%. These results bring some evidence that the test suite may be extended at model level so that it has a positive impact on code coverage. The delta (Δ) coverage represents the difference in line coverage between the original project and the extended model.

ID	Participant	#Abstract Events	#Concrete Events	% of Concrete Events	Effort Time to Concretize Abstract Methods (min)	% of Line Coverage Original Project	% of Line Coverage Extended Model	Δ Coverage	% of Coverage Increased
1	A	10	80	88.89%	10	5.51%	6.04%	0.53%	9.62%
	B	1	170	99.42%	3	4.69%	5.76%	1.07%	22.81%
2	C	24	128	84.21%	29	6.37%	7.51%	1.14%	17.90%
	D	21	56	72.73%	28	6.98%	7.53%	0.55%	7.88%
3	E	59	73	55.30%	52	27.31%	34.84%	7.53%	27.57%
	F	21	59	73.75%	29	30.95%	34.81%	3.86%	12.47%
4	G	64	83	56.46%	62	7.05%	10.71%	3.66%	51.91%
	H	9	102	91.89%	13	9.55%	10.30%	0.75%	7.85%
5	I	50	74	59.68%	67	25.22%	31.66%	6.44%	25.54%
	J	28	40	58.82%	37	28.44%	31.50%	3.06%	10.76%
6	K	9	61	87.14%	13	41.24%	42.89%	1.65%	4.00%
	L	40	86	68.25%	26	37.95%	44.10%	6.15%	16.21%
7	M	32	81	71.68%	30	9.26%	9.83%	0.57%	6.16%
	N	26	50	65.79%	27	1.98%	9.78%	7.80%	393.94%
8	O	27	94	77.69%	23	15.14%	17.20%	2.06%	13.61%
	P	26	25	49.02%	28	15.20%	17.34%	2.14%	14.08%
9	Q	63	70	52.63%	50	5.36%	7.71%	2.35%	43.84%
	R	61	79	56.43%	55	6.67%	7.69%	1.02%	15.29%
Average:				**70.54%**					**38.97%**
Minimum:				**49.02%**					**4.00%**
Maximum:				**99.42%**					**393.94%**

Table 4.4: Data About Test Case Generation

According to Table 4.5, the projects had a greater increase in LoC during the automatic generation of test cases. After that, in the concretization of the abstract test cases, the projects did not increase so much, showing that most of the source code for the new test cases was automatically generated. Also, we observed that most of the changes in the concretization step occurred in the PageObjects classes. The test classes, on the other hand, did not require many changes, as the tool generated them with a certain accuracy. In 9 projects, the test classes remained with the same LoC metric after the concretization step. In some projects, the number of methods in the PageObjects classes has increased between the generation of test cases and concretization step. This happened because, in the concretization step, some auxiliary methods were created to maintain the code. In project A, of application #1, the original project had 1483 lines of code. After generating the test cases with the tool, the project got 1837 lines of code (354 automatically generated LoC) and 1895 after concretizing the abstract PageObject methods (58 new lines implemented by a developer). So, 85.92% of the new lines were automatically generated by the tool. In the worst scenario, in the participant's project A, of application #9, 64.33% were automatically generated, which is a good scenario yet. If we consider only the generated test classes (without new PageObjects methods and classes), in the worst scenario, 95.77% of the lines of code were automatically generated.

ID	Participant	Metrics												
		#Test Cases		Project LoC			Tests LoC			#PO Classes		#PO Methods		
		1. Original Project	3. After Concretization	1. Original Project	2. Model After Extension	3. After Concretization	1. Original Project	2. Model After Extension	3. After Concretization	1. Original Project	3. After Concretization	1. Original Project	2. Model After Extension	3. After Concretization
1	A	16	22	1483	1837	1895	657	978	984	15	18	112	125	125
	B	10	23	1565	2211	2224	517	1137	1137	18	18	151	152	155
2	C	10	18	1217	1803	1930	520	983	985	12	14	81	119	120
	D	10	18	687	1227	1362	97	512	512	14	15	73	108	108
3	E	10	22	777	1670	1913	160	790	790	10	15	84	154	154
	F	12	21	1451	2001	2192	537	940	946	17	20	135	175	175
4	G	10	21	1125	1994	2298	440	1015	1015	13	26	90	171	171
	H	13	21	1186	1743	1809	314	796	810	24	28	115	130	132
5	I	10	19	1237	1962	2276	477	943	943	16	26	82	148	149
	J	10	19	821	1354	1507	86	479	479	21	26	71	108	108
6	K	10	16	563	915	989	112	416	419	5	5	62	74	75
	L	10	18	525	1068	1196	154	591	591	7	7	33	66	66
7	M	10	19	844	1432	1728	325	776	788	9	11	70	107	107
	N	11	21	1228	1740	1844	377	778	777	7	10	75	107	107
8	O	10	19	699	1299	1441	170	646	667	15	22	68	94	96
	P	10	18	484	910	1033	138	430	430	11	17	43	77	77
9	Q	10	20	1378	2242	2721	221	788	789	43	52	124	192	192
	R	10	19	1213	2061	2377	153	730	730	34	41	96	169	169

Table 4.5: Projects metrics at each step

4.3 DISCUSSION

The experimental evaluation validated the proposed approach for the use of MBT in existing test suite for web applications. The results obtained provide evidence on the feasibility of applying the approach and the benefits that can be achieved by adopting it. During the execution of the evaluation, some considerations were noted and are discussed as follows.

It is possible to observe that the use of the approach is feasible in real applications, with potential gains in line coverage with the reuse of the test suites that were used in the experiment. As discussed, of the new events generated after the extension of the models, on average 70.54% are already concretized methods, which facilitated the extension of the model, as it does not require the creation of the abstract methods in the tool, and represents a shortcut in test case generation.

Regarding the execution time to infer the model and generate the new test cases, the approach was also viable, since the growth in relation to the size of the projects seems to be linear.

Without using MoLeWe, we can argue that the time to generate new test cases would be longer, since the tester would have to manually generate the large portion of code generated automatically by the tool. It is important to remember that the use of the model, for generating new test cases, brings other advantages, such as the ease in traceability of the tests, allows the automatic generation of test cases, and the validation of the SUT requirements (UTTING; LEGEARD, 2006).

Due to the readability of the models brought by the PageObjects pattern, in model extension step we hardly needed to read the project source code to understand how test cases work, because understanding the model was almost always enough. Some limitations were found in the use of inheritance, as the tool does not support it.

The task of extending the model depends on how well the PageObjects are structured, as the creation of a new test case should fit the previously created structure and should avoid duplicate PageObjects methods. The smaller the granularity of the PageObjects methods, the better the reuse of the methods, as it is not necessary to create new methods that partially perform the actions of an already implemented method.

Although MoLeWe supports only test suites developed in Java, this approach can be replicated for any other language. However, to maintain the readability of the generated model, it is important that there is an isolation of the functions that manipulate the web pages in abstraction classes, as done in the PageObjects pattern.

4.4 THREATS TO VALIDITY

One limitation of the proposed approach and MoLeWe is the assumption of PageObject-based test cases. While the tool can be extended to support test suites implemented using different designs, PageObjects are widely utilized in web test automation and there are tools that, with high accuracy, can generate automatically PageObject classes (STOCCO et al., 2017). Besides, the correct adoption of PageObjects can improve the development, reuse and maintenance of test cases.

A potential threat is that the sample of web applications may not be representative. In this case, it is important to emphasize that the Selenium-based test cases are executed in front-end, interacting only with technologies like HTML, CSS, and Javascript. So, the back-end language does not have so much influence in test execution. A benefit was that we could compare the code coverage without the threat of dealing with different programming languages. To minimize the threats on the sample, we selected applications from different domains and with varied project sizes.

The participants had a good knowledge of web application development and testing. Nevertheless, the inclusion of participants with a diversity of skills and levels of technology proficiency might produce different results. Regarding the outcomes, the selected SUTs are mature and, as a consequence, the results on fault detection were limited.

As for the experimental study, a limitation is that the approach has not been compared to any other present in the literature. This is a preliminary study and other related approaches do not have a specific focus on test suites developed with the Java language, with the PageObjects pattern, making the comparison difficult.

4.5 FINAL REMARKS

This chapter presented an experimental study to evaluate the proposed approach of using MBT in projects that already have a test suite implemented in Java with the PageObjects pattern. Eighteen participants developed test projects for nine web applications, which were inserted as input in the tool. The results provided evidence that the approach is feasible, and that it is possible to increase the line coverage and create new test cases taking advantage of the PageObjects classes already implemented in the existing test suite. The next chapter concludes this thesis, raising the main contributions, limitations and possible future work.

5 CONCLUSION

With agile methodologies, the testing activity is also a responsibility of the developers. In this scenario, testers may encounter scenarios where a test suite may be leveraged to derive new test cases from it. In this context, there is a lack of MBT tools that support the generation of new test cases from test suites implemented in common languages, like Java.

This work presented an approach to improve the synergy between existing test cases and model-based testing. This approach involved the model inference from a test suite repository with automated tests, test extensions by a human tester at model level, and abstract and concrete generation of new tests from the extended model. In this study, we focused on web applications with end-to-end GUI automated tests, using an event-driven technique to represent test models. A tool, named MoLeWe, was developed to support the approach' steps of model inference, extension, and test generation. This approach uses the PageObjects pattern as an abstraction layer to make the model readable.

We evaluated the approach with 18 participants, where each one implemented a test suite for an open source web application. These projects were used as input to the tool and the inferred models were extended, considering different test cases implemented by the participants. The results gave evidence that the proposed approach is feasible and scalable, while presenting a reasonable cost-effectiveness. The new test cases generated from the extended model increased the line coverage by 38.97%, on average.

5.1 CONTRIBUTIONS

The main contribution of this thesis was the definition of an approach that allows a greater engagement between MBT and projects that already have an existing test suite. The approach made it possible to take advantage of a current test suite in the generation of new test cases, allowing the reuse of the suite and reaching larger portions of line coverage, together with the advantages of the MBT approach.

As a secondary contribution, this work presented the MoLeWe tool to validate the proposed approach. The tool was developed in Java and allowed the simulation of all steps of the approach and an experimental study that evaluated the feasibility of the proposal. Thus, this work paves the way for solutions in which teams that have an existing test suite can use MBT to generate new test cases, reusing the tests already implemented.

5.2 LIMITATIONS AND FUTURE DIRECTIONS

This work does not completely solve the problem of pragmatic application of MBT in web projects that already have an existing test suite, but it presents a small scale direction that can be evolved and adapted according to the context found. One of the limitations is the limited scope of MoLeWe, which requires Java test projects implemented with PageObjects. However, the results are promising and motivate replications with other contexts and technologies.

In future work, we intend to extend the approach to use other modeling techniques like state machines. Other direction is to formalize the relation between the test model and PageObjects, employing more powerful model inference techniques. Different model coverage criteria (like edge-pair and prime path coverage) could be investigated to produce more effective test suites.

To increase the approach applicability beyond PageObject-based test suites, we plan to insert a step in MoLeWe of structured extraction, where PageObjects are automatically identified and extracted from existing code. This allows to the usage of MoLeWe even in test projects that have no PageObjects. The algorithm was designed (see Appendix A), however its implementation remains a future work.

In order to adapt the tool to the current agile development methodologies, we intend to investigate how the approach can be used with continuous integration tools.

To generate test cases that cover new elements, MoLeWe employs an algorithm focused on identifying the shortest paths in the model. However, different properties may be taken into account, besides generating a short test case. For instance, the tool may target improved code coverage or higher coverage of the model.

As this is the first evaluation of the approach, more experimental studies are required to clarify its benefits and drawbacks. We plan to investigate and characterize how practitioners extend their web test suites. From this, we can compare MoLeWe with such practices and tailor a more efficient and effective tool. Test cases of open source applications could be used to assess the diversity of existing test suites and their suitability for the approach. Finally, replications of this study with more and different participants, other web applications, and different experimental settings would be desirable.

REFERENCES

ADAMSEN, C. Q.; MEZZETTI, G.; MØLLER, A. Systematic execution of android test suites in adverse conditions. In: ACM. **Proceedings of the 2015 International Symposium on Software Testing and Analysis**. 2015. p. 83–93.

AICHERNIG, B. K.; BRANDL, H.; JÖBSTL, E.; KRENN, W.; SCHLICK, R.; TIRAN, S. Killing strategies for model-based mutation testing. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 25, n. 8, p. 716–748, 2015.

ANAND, S.; BURKE, E. K.; CHEN, T. Y.; CLARK, J.; COHEN, M. B.; GRIESKAMP, W.; HARMAN, M.; HARROLD, M. J.; MCMINN, P.; BERTOLINO, A. et al. An orchestrated survey of methodologies for automated software test case generation. **Journal of Systems and Software**, Elsevier, v. 86, n. 8, p. 1978–2001, 2013.

ANGLUIN, D. Inference of reversible languages. **J. ACM**, ACM, New York, NY, USA, v. 29, n. 3, p. 741–765, jul. 1982. ISSN 0004-5411.

ARANTES, A.; JÚNIOR, V. S.; VIJAYKUMAR, N.; SOUZA, E. Tool support for generating model-based test cases via web. **Int. J. of Web Engineering and Technology**, v. 9, p. 62 – 96, 01 2014.

BARR, E. T.; HARMAN, M.; MCMINN, P.; SHAHBAZ, M.; YOO, S. The oracle problem in software testing: A survey. **IEEE Transactions on Software Engineering**, v. 41, n. 5, p. 507–525, 2015.

BECK, K. L. **Test-driven Development - by example**. : Addison-Wesley, 2003. (The Addison-Wesley signature series). ISBN 978-0-321-14653-3.

BELLI, F.; ENDO, A. T.; LINSCHULTE, M.; SIMAO, A. A holistic approach to model-based testing of web service compositions. **Software: Practice and Experience**, Wiley Online Library, v. 44, n. 2, p. 201–234, 2014.

BERNARDINO, M.; RODRIGUES, E. M.; ZORZO, A. F.; MARCHEZAN, L. Systematic mapping study on mbt: tools and models. **IET Software**, IET, v. 11, n. 4, p. 141–155, 2017.

BIAGIOLA, M.; RICCA, F.; TONELLA, P. Search based path and input data generation for web application testing. In: SPRINGER. **International Symposium on Search Based Software Engineering**. 2017. p. 18–32.

BIAGIOLA, M.; STOCCO, A.; RICCA, F.; TONELLA, P. Diversity-based web test generation. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. 2019. p. 142–153.

BUDNIK, C. J. **Test generation using event sequence graphs**. Tese (Doutorado) — University of Paderborn, Germany, 2006.

CHELIMSKY, D.; ASTELS, D.; HELMKAMP, B.; NORTH, D.; DENNIS, Z.; HELLESOY, A. **The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends**. 1st. ed. : Pragmatic Bookshelf, 2010. ISBN 1934356379.

CHOUDHARY, S. R.; GORLA, A.; ORSO, A. Automated test input generation for Android: Are we there yet? In: **Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. Washington, DC, USA: , 2015. p. 429–440. ISBN 978-1-5090-0025-8.

CHRISTOPHE, L.; STEVENS, R.; ROOVER, C. D.; MEUTER, W. D. Prevalence and maintenance of automated functional tests for web applications. In: IEEE. **Proceedings of 30th IEEE International Conference on Software Maintenance and Evolution**. 2014. p. 141–150.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms, Third Edition**. 3rd. ed. : The MIT Press, 2009. ISBN 0262033844.

DANGLOT, B.; VERA-PEREZ, O.; YU, Z.; ZAIDMAN, A.; MONPERRUS, M.; BAUDRY, B. A snowballing literature study on test amplification. **Journal of Systems and Software**, v. 157, 2019.

DANGLOT, B.; VERA-PÉREZ, O. L.; BAUDRY, B.; MONPERRUS, M. Automatic test improvement with DSpot: A study with ten mature open-source projects. **Empirical Software Engineering**, Springer, p. 1–33, 2019.

DIXIT, R.; LUTTEROTH, C.; WEBER, G. FormTester: Effective integration of model-based and manually specified test cases. In: **Proceedings of 37th IEEE/ACM International Conference on Software Engineering, ICSE**. 2015. v. 2, p. 745–748. ISSN 0270-5257.

ENDO, A. T. **Model based testing of service oriented applications**. Tese (Doutorado) — São Paulo University, 2013.

ENTIN, V.; WINDER, M.; ZHANG, B.; CHRISTMANN, S. Combining model-based and capture-replay testing techniques of graphical user interfaces: An industrial approach. In: IEEE. **Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation, ICST**. 2011. p. 572–577.

ENTIN, V.; WINDER, M.; ZHANG, B.; CHRISTMANN, S. Introducing model-based testing in an industrial Scrum project. In: IEEE PRESS. **Proceedings of the 7th International Workshop on Automation of Software Test**. 2012. p. 43–49.

ENTIN, V.; WINDER, M.; ZHANG, B.; CLAUS, A. A process to increase the model quality in the context of model-based testing. In: IEEE. **Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation, ICST**. 2015. p. 1–7.

FARD, A. M.; MIRZAAGHAEI, M.; MESBAH, A. Leveraging existing tests in automated test generation for web applications. In: ACM. **Proceedings of the 29th ACM/IEEE international conference on Automated software engineering**. 2014. p. 67–78.

FARTO, G. C.; ENDO, A. T. Reuse of model-based tests in mobile apps. In: ACM. **Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES)**. 2017. p. 184–193.

FOWLER, M. **PageObject**. set. 2013. Available on: <https://martinfowler.com/bliki/PageObject.html>.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. **Refactoring: Improving the Design of Existing Code**. Pearson Education, 2012. (Addison-Wesley Object Technology Series). ISBN 9780133065268. Available on: <https://books.google.com.br/books?id=HmrDHwgkbPsC>.

FRASER, G.; ZELLER, A. Generating parameterized unit tests. In: **Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)**. New York, NY, USA: , 2011. p. 364–374. ISBN 978-1-4503-0562-4.

GAROUSI, V.; ELBERZHAGER, F. Test automation: not just for test execution. **IEEE Software**, IEEE, v. 34, n. 2, p. 90–96, 2017.

GAROUSI, V.; FELDERER, M. Worlds apart: Industrial and academic focus areas in software testing. **IEEE Software**, IEEE, n. 5, p. 38–45, 2017.

KRAMER, A.; LEGEARD, B. **Model-Based Testing Essentials-Guide to the ISTQB Certified Model-Based Tester: Foundation Level**. : John Wiley & Sons, 2016.

KRAMER, A.; LEGEARD, B.; BINDER, R. V. **2016/2017 Model-based Testing User Survey**. 2017. Available on: <http://www.cftl.fr/wp-content/uploads/2017/02/2016-MBT-User-Survey-Results.pdf>.

LEOTTA, M.; CLERISSI, D.; RICCA, F.; SPADARO, C. Improving test suites maintainability with the page object pattern: An industrial case study. In: IEEE. **Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. 2013. p. 108–113.

LI, N.; ESCALONA, A.; KAMAL, T. Skyfire: Model-based testing with Cucumber. In: IEEE. **2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. 2016. p. 393–400.

MAO, K.; HARMAN, M.; JIA, Y. Sapienz: Multi-objective automated testing for Android applications. In: **Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)**. New York, NY, USA: ACM, 2016. p. 94–105. ISBN 978-1-4503-4390-9.

MAYES, N. **Software Testing Spends to Hit EUR100BN by 2014**. 2010.

MESBAH, A. Software analysis for the web: Achievements and prospects. In: IEEE. **Proceedings of the IEEE 23rd International Conference On Software Analysis, Evolution, and Reengineering (SANER)**. 2016. v. 5, p. 91–103.

MESBAH, A.; DEURSEN, A. van; LENSELINK, S. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. **ACM Trans. Web**, ACM, New York, NY, USA, v. 6, n. 1, p. 3:1–3:30, mar. 2012. ISSN 1559-1131.

MICSKEI, Z. **Code-based test generation**. dez. 2015. Available on: <http://mit.bme.hu/~micskeiz/pages/code_based_test_generation.html>.

MICSKEI, Z. **Model-based testing (MBT)**. set. 2017. Available on: <http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html>.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. : John Wiley & Sons, 2011.

PAWLAK, R.; MONPERRUS, M.; PETITPREZ, N.; NOGUERA, C.; SEINTURIER, L. SPOON: A library for implementing analyses and transformations of java source code. **Softw. Pract. Exp.**, v. 46, n. 9, p. 1155–1179, 2016. Available on: <https://doi.org/10.1002/spe.2346>.

RAFI, D. M.; MOSES, K. R. K.; PETERSEN, K.; MÄNTYLÄ, M. V. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: IEEE PRESS. **Proceedings of the 7th International Workshop on Automation of Software Test**. 2012. p. 36–42.

SAMS, P. **Selenium essentials**. : Packt Publishing Ltd, 2015.

SCHULZE, C.; GANESAN, D.; LINDVALL, M.; CLEAVELAND, R.; GOLDMAN, D. Assessing model-based testing: an empirical study conducted in industry. In: ACM. **Companion Proceedings of the 36th International Conference on Software Engineering**. 2014. p. 135–144.

SCHULZE, C.; LINDVALL, M.; BJORGVINSSON, S.; WIEGAND, R. Model generation to support model-based testing applied on the NASA DAT web-application - An experience report. In: **Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)**. 2015. p. 77–87.

SCHUR, M.; ROTH, A.; ZELLER, A. Mining behavior models from enterprise web applications. In: **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering**. : ACM, 2013. p. 422–432. ISBN 978-1-4503-2237-9.

SLACK, J. M. Modeltester: a tool for teaching model-based testing. **Journal of Computing Sciences in Colleges**, Consortium for Computing Sciences in Colleges, v. 27, n. 1, p. 37–46, 2011.

SPRING.IO. **Spring PetClinic Sample Application**. 2014. Available on: <https://github.com/spring-projects/spring-petclinic>.

STOCCO, A.; LEOTTA, M.; RICCA, F.; TONELLA, P. Why creating web page objects manually if it can be done automatically? In: IEEE PRESS. **Proceedings of the 10th International Workshop on Automation of Software Test**. 2015. p. 70–74.

STOCCO, A.; LEOTTA, M.; RICCA, F.; TONELLA, P. APOGEN: Automatic page object generator for web testing. **Software Quality Journal**, v. 25, n. 3, p. 1007–1039, Sep 2017. ISSN 1573-1367.

SU, T.; MENG, G.; CHEN, Y.; WU, K.; YANG, W.; YAO, Y.; PU, G.; LIU, Y.; SU, Z. Guided, stochastic model-based GUI testing of Android apps. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. : ACM, 2017. (ESEC/FSE 2017), p. 245–256. ISBN 978-1-4503-5105-8.

THUMMALAPENTA, S.; MARRI, M. R.; XIE, T.; TILLMANN, N.; HALLEUX, J. de. Retrofitting unit tests for parameterized unit testing. In: SPRINGER. **International Conference on Fundamental Approaches to Software Engineering**. 2011. p. 294–309.

TILLMANN, N.; SCHULTE, W. Unit tests reloaded: Parameterized unit testing with symbolic execution. **IEEE software**, IEEE, v. 23, n. 4, p. 38–47, 2006.

TORENS, C.; EBRECHT, L.; LEMMER, K. Inverse model based testing–generating behavior models from abstract test cases. In: IEEE. **Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. 2011. p. 75–78.

TORENS, C.; EBRECHT, L.; LEMMER, K. Starting model-based testing based on existing test cases used for model creation. In: **Proceedings of the 2011 IEEE 11th International Conference on Computer and Information Technology**. 2011. p. 320–327.

UTTING, M.; LEGEARD, B. **Practical Model-Based Testing: A Tools Approach**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. 3-59 p. ISBN 0123725011.

WIKLUND, K.; ELDH, S.; SUNDMARK, D.; LUNDQVIST, K. Impediments for software test automation: A systematic literature review. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 27, n. 8, p. e1639, 2017.

XIE, T.; NOTKIN, D. Tool-assisted unit-test generation and selection based on operational abstractions. **Automated Software Engineering**, Springer, v. 13, n. 3, p. 345–371, Jul 2006.

XU, D.; TU, M.; SANFORD, M.; THOMAS, L.; WOODRASKA, D.; XU, W. Automated security test generation with formal threat models. **IEEE transactions on dependable and secure computing**, v. 9, n. 4, p. 526–540, 2012.

ZANDER, J.; SCHIEFERDECKER, I.; MOSTERMAN, P. J. **Model-Based Testing for Embedded Systems**. 1st. ed. : CRC Press, Inc., 2011.

ZECH, P.; FELDERER, M.; KALB, P.; BREU, R. A generic platform for model-based regression testing. In: SPRINGER. **International Symposium On Leveraging Applications of Formal Methods, Verification and Validation**. 2012. p. 112–126.

ZHANG, P.; ELBAUM, S. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 32:1–32:28, set. 2014. ISSN 1049-331X.

APPENDIX A – ALGORITHM FOR GENERATING PAGEOBJECTS

Algorithm 3 describes a refactoring procedure to obtain a clear separation between test cases and test abstractions from a set of classes with test cases. First, *Extract Method* (FOWLER et al., 2012) is applied to isolate specific framework calls (Lines 1-6). *Extract Class* (FOWLER et al., 2012) moves them to a different class (so-called *adapter*); it also instruments the *adapter* class to collect the page in which a given method is called (Lines 7-12). The test cases are run (Line 13) and the collected information is used to create the PageObject classes (Lines 14-21).

Algorithm 3: Generating PageObjects

```
   input  : Test classes $TCl$
   output: Refactored test classes $TCl$
   /* isolate test framework calls in internal methods */
1  foreach $clazz \in TCl$ do
2      foreach $m \in methods(clazz)$ do
3          identify set of blocks $B$ to be extracted of $m$
4          apply extract method for blocks $B$
5      end
6  end
   /* extract adapter class and instrument it */
7  foreach $clazz \in TCl$ do
8      apply extract class to $clazz$
9      create class $adapter$ with methods extracted
10     add $adapter$ to set $ADP$
11     instrument $adapter$ to collect page info
12 end
13 run tests in $TCl$
   /* divide adapter classes in POs */
14 foreach $adapter \in ADP$ do
15     foreach $m \in methods(adapter)$ do
16         retrieve $Pagename$ of $m$
17         create class $PagenamePO$ if not exists
18         apply extract method $m$ from $adapter$ to $PagenamePO$
19     end
20     delete class $adapter$
21 end
```