Teasy: A domain-specific language to reduce time and facilitate the creation of tests in web applications

Yury Alencar Lima[1], **Elder de Macedo Rodrigues**[1], **Fabio Paulo Basso**[1], **Rafael A. P. Oliveira**[2]

[1]Universidade Federal do Pampa (UNIPAMPA)
Código Postal 97.546-550 – Alegrete – RS – Brasil

yuryalencar19@gmail.com, fabiobasso@unipampa.edu.br

elderrodrigues@unipampa.edu.br

[2]Universidade Tecnológica Federal do Paraná (UTFPR)
Código Postal 85.660-000 – Dois Vizinhos – PR – Brasil

raoliveira@utfpr.edu.br

Abstract. *Software testing automation is one of the most challenging activities in Software Engineering scenarios. Moden-Based Testing (MBT) is a feasible strategy to alleviate efforts on automating testing activities. Trough a model that specifies the behavior of the Software Under Testing (SUT), MBT approaches are useful strategies to generate test cases and run them. However, some domains such as, web applications require extra efforts on applying MBT approaches. Due to this, in this study we propose and validate Teasy - a Domain Specification Language (DSL) that makes MBT feasible for web application. Through the conduction of a Proof-of-Concept on testing a real-world web application, we noticed Teasy has potential to evolve to effectively support software development environments. Using a real-world application and projects with manually seeded faults, Teasy testing scenarios have detected 78,57% of the functional inconsistencies.*

Resumo. *A automação de teste de software é uma das atividades mais desafiadoras nos cenários de Engenharia de Software. Teste Baseado em Modelos (MBT) é uma estratégia viável para aliviar os esforços na automação das atividades de teste. Através de um modelo que especifica o comportamento do Software Under Testing (SUT), as abordagens MBT são estratégias úteis para gerar casos de teste e executá-los. No entanto, alguns domínios, como aplicativos da web, exigem esforços extras na aplicação de abordagens MBT. Por isso, neste estudo propomos e validamos o Teasy - uma Domain Specification Language (DSL) que torna o MBT viável para aplicação web. Por meio da realização de uma Prova de Conceito para testar um aplicativo da web do mundo real, percebemos que o Teasy tem potencial para evoluir para oferecer suporte eficaz aos ambientes de desenvolvimento de software. Usando um aplicativo do mundo real e projetos com falhas semeadas manualmente, os cenários de teste do Teasy detectaram 78,57 % das inconsistências funcionais.*

1. Introduction

Software Engineering (SE) is the wide usage of engineering resources in software projects [Pressman 2010]. Analysis, projects, measurements (metrics), testing, reviews, refactoring, and verification are some of the engineering activities promoted by SE routines. All of the SE arrangements have one common goal: building a software product with quality [Sommerville 2010]. The result of using SE concepts properly brings

five main characteristics to the final software product: *(i)* maintainability, *(ii)* precision, *(iii)* productivity, *(iv)* functionality, and *(v)* longevity. Neglecting SE activities lead projects to be over budget, late, missing functions or a combination of all three aspects [Sommerville 2010, Pressman 2010].

Practitioners know that applying SE activities in software projects brings up a well-know "dilemma" related to cost aspects. SE activities are necessary to make the quality sure in final software products. However, SE activities need for several human labor efforts on analysis, modeling, verification, testing, and validation, which makes the project to be costly. Finally, SE activities are time-consuming and non-trivial to conduct [Sommerville 2010].

An accepted solution to alleviate project costs due to the application of SE activities is to implement their automation (or semi-automation) [Bertolino 2007]. Automate SE activities trough scripts, tools or CASE (*Computer-Aided Software Engineering*)-based systems reduces the development time, allowing the quality of the final product and adapting the customer to market competitiveness [Bertolino 2007].

Regarding SE activities, Software testing is one of the most challengers activities to automate [Bertolino 2007]. Generating valid and invalid inputs, calculate expected outputs, and providing reports are some of the desirable activities from an automate testing scenario. Further, depending on the application domain, automating testing activities is even harder.

An instance of non-trivial domain to automate test is the web applications. This domain has some technical particularities that make the complete testing automation costly. Requests, responses, services, events, protocols, security, and internet issues are key-elements that makes web applications harder to test than regular desktop-applications [Vani et al. 2013].

Model-Based Testing (MBT) can be a viable alternative to alleviate problems related to testing automation of web applications. MBT can be used to perform the automated generation of tests, using predefined models, thus improving the efficiency on creating test cases and reducing project time. MBT allows practitioners to update only one model (based on the system specification) and as consequence all of test cases can be generated again and updated. This feature improves the test cases' maintainability and it reduces the effort required [Utting and Legeard 2010]. However, using MBT is not trivial and it can be complex even for expert domain users. An alternative on using the MBT approach is through a *Domain-Specific Language* (DSL). DSLs are languages that aim to represent a specific domain, increasing the productivity and engagement of those involved, enabling or not, to generate artifacts since created for this purpose, and consequently reducing the complex to domain expert users [Fowler 2010]. For instance, using a DSL to apply MBT in the web application testing process facilitates the communication between development and quality teams, reducing the effort on creating automated tests.

This study presents a DSL called Teasy that provides an approach on using MBT for web application. Created in the context of the final project of an undergraduate course (Bachelor of Software Engineering), Teasy is a DSL focused on three main concepts: *(i)* code reuse, *(ii)* smooth maintenance of testing routines, and *(iii)* easy of usage. Finally, to provide a bird-eye-view of the process of using Teasy, we provide a validation using a

real-world application.

This study brigs three main contributions to state-of-the-art: *(i)* a comprehensive DSL to automate MBT of web applications;*(ii)* an useful process to generate test scripts using Teasy; and *(iii)* a massive discussion towards on bridging the gaps between theory and practice of MBT and contemporary software development projects.

The rest of this paper is organized as follows: Section 2 that presents background with the technical concepts to understand how Teasy was developed. Section 3 introduces Teasy, its general structure, and a process for creating an automated test using the language. Section 4 presents a proof-of-concept on using Teasy to test a real-world web application taken from a public repository. Section 5 presents an analysis and results of the validation presented in Section 4. Section 6 presents the related works associated to our study. Finally, in Section 7 we present some final remarks containing a brief of our results, study contributions, and some future work.

2. Background

This section presents essential concepts to fully understand the Teasy process and its goals.

2.1. Functional Tests

Testing activities are intended to detect faults in software products [Delamaro et al. 2013]. This practice positively impacts product-related reliability and it assists to ensure that the software product meets its requirements. Functional tests represent a testing method in which the system is taken as a black-box, that is, the application's source codes are not considered, only its behavior [Delamaro et al. 2013]. This behavior is, then, derived from the software requirements and verified through test cases. Thus, enabling the detection of functional faults before the product is delivered. In this way it is possible to reduce the costs related to the need for maintenance in the application after its delivery, making bigger the life cycle of the software product [Delamaro et al. 2013].

2.2. Model-Based Testing

The *Model-Based Test* (MBT) aims to increase the level of abstraction necessary for the development of the tests [Utting and Legeard 2010]. Providing a better understanding of the software to be tested and improving its specification. In this approach, a model is defined through the SUT's (*Software under Testing*) behaviors and requirements. Using these information, test cases, and/or scripts of automated tests are generated. A model is a description of a SUT's behavior.

MBT mitigates possible ambiguities related to textual documentation and can reduce costs, given that faults can be detected even before the product is developed. This is possible because this approach requires only the model related to the software to perform the verification. In addition, MBT also allows non-programmers to create automated tests, due to the possibility of automatic generation of test scripts [Utting and Legeard 2010].

2.3. Domain-Specific Languages

Unlike *General Purpose Languages* (GPL) like JavaScript, Java, among others, *Domain-Specific Languages* (DSL) are computer languages that have the objective of supporting

and models for problems associated to a given domain [Mernik et al. 2005]. Then, a given DSL goals to solve a specific domain problem. Since DSLs cannot solve adversities that occur in others scenarios/situations, they are considered part of the Engineering Domain [Fowler 2010]. Based on this, its level of abstraction is usually higher than a GPL, making it easier for domain experts to use DSLs on solving problems [Živanov et al. 2008].

Even though a DSL can increase the productivity and engagement of domain experts, its implementation is not a trivial task. In addition, the profile of the specialists who will use the language can influence its type and way of development. Based on this, there several ways to implement a DSL that influence the definition of its types. Considering the DSLs' essence, [Fowler 2010] present the following taxonomy for DSLs:

- **Internal Languages:** Internal or Embedded Languages are defined and implemented within another existing language that is usually a GPL. As they are related to the GPL in which they were created, domain experts who will use them have an advantage if they have prior knowledge related to the respective GPL;
- **External Languages:** External Languages are DSLs independent of other languages such as GPLs, they contain their own analyzer and generator when they have; e
- **Languages Workbenches:** The *Languages Workbenches* (LW) are tools that support the implementation of DSL, providing a development environment. This environment provides a scheme for defining a semantic model, editing the DSL and performing behavioral semantics through interpretation and code generation if necessary.

3. Teasy

In this section, we present the Teasy language with its structure and all process necessary to create functional automated tests. Teasy allows testers to create an automated test using just five setting files, reducing the effort and complexity on applying MBT.

Teasy is designed to provide the following benefits: *(1)* Better project organization; *(2)* Reducing the complexity of the project; *(3)* Allowing the adoption of a design pattern for testing; *(4)* Automated generation of functional test scripts; *(5)* Reduction of the effort required to create the test scripts due to the fact that Teasy has a systematic data projection of the data, requiring from testers only some key-elements; *(6)* generation of reports in navigable HTML (*HyperText Markup Language*); *(7)* Allowing the generation of test sequences based on the paging files; *(8)* Providing a comprehensive development environment, with suggestions, highlights, and auto completes; *(9)* Providing a effortless maintenance of test codes; and *(10)* Promote code reusing, reducing the time required to create an automated test set.

3.1. General Structure

The DSL promoted by Teasy contains a simple structure based on page object pattern providing easy maintainability. In addition, Teasy contains only five archive for creating functional automated testing scripts.

Teasy's file structure follows some specific files: **Configuration:** in this file, information related to the execution configuration is inserted, such as the browser to use, implicit timeout, and base URL (*Uniform Resource Locator*) for the web application;

Components: in this file, the page components are registered, using their unique name with a selector defined using the HTML of the web application; **Page:** in this file, the pages are registered with their actions, each action make manipulations in components registered in the components file; **PageRegister:** after a page is finalized, it is necessary to register that page name in this file for generation code import to work; and **Flows:** on flows file, it is possible to create test scenarios using the actions registered in page files.

Teasy is designed to reduce the time required to write testing code, providing automatically repetitive information, and requiring only important information. In addition, Teasy's environment provides auto completes and highlights for a better users/tester experience.

Regarding the testing code generated, it's important to highlight that Teasy's codes are not executable. However the code generated by Teasy can be executed using the Robot framework [Bisht 2013] and Python. Due to that, it also contains other benefits provided in that framework, how a report generation.

The generated code structure contains: *(1)* components' directory and representation of components registered in Teasy code; *(2)* "config" directory containing all of the information related of the configuration and page register of Teasy files; *(3)* pages' directory including all of the pages created in page files; *(4)* commons directory containing a hooks file generated by Teasy for open and close browser; and *(5)* finally, tests directory refers to all testing scenarios created in the flow files.

3.2. Teasy process

To create a test suite using the Teasy language, a set of implemented pages or a set of page prototypes is required, Figure 1 shows a BPMN (*Business Process Model and Notation*) diagram of this process.

The first step in creating tests using Teasy. This step aims at mapping all of the pages of the web application that will be involved in the tests. Additionally, this step verifies whether all of the components of the pages are mapped in the component file. A component is an HTML element presented on a page, this element can be a button, input field, among others. If there are unmapped components, tester can insert it in the component file as one can see in Figure 2.

Once all of the page components are mapped, tester are required to create page files. For each page in the application, it is necessary to create a page file on Teasy. In this file, page actions are registered, such as click on some button, insert text on the field, among others. Figure 3 presents an example of this implementation.

With registered pages, it is possible to create test scenarios using flow files. In this file, the page's actions are inserted in sequence to create a flow. This string must contain actions from the first page of the application, and the first page must be accessed in the base URL informed in the configuration file. Figure 4 presents an example of the implementation of this file.

To compile the tests, it is necessary to implement a simple configuration. For each page created, its respective name must be registered in the PageRegister file and the configuration file must also be filled out. As shown in Figure 5 and 6 respectively. The last step is to get all the generated files.

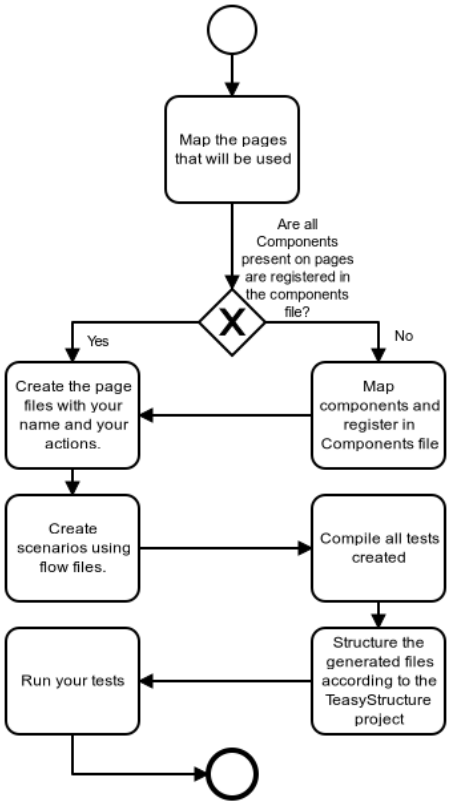

Figure 1. Teasy Process

```
SYSTEM COMPONENTS

  INSERT NAME COMPONENT: AC_BUTTON
  INSERT SELECTOR: id
  INSERT SELECTOR VALUE: AC

  INSERT NAME COMPONENT: BUTTON_0
  INSERT SELECTOR: id
  INSERT SELECTOR VALUE: 0

  INSERT NAME COMPONENT: BUTTON_1
  INSERT SELECTOR: id
  INSERT SELECTOR VALUE: 1
```

Figure 2. Teasy – component file

```
PAGE NAME: CalculatorHome

  ACTION: Click Button 0 $<
    CLICK ELEMENT:  BUTTON_0

  >$
  ACTION: Click Button 1 $<
    CLICK ELEMENT:  BUTTON_1

  >$
  ACTION: Click Button 2 $<
    CLICK ELEMENT:  BUTTON_2

  >$
```

Figure 3. Teasy – page file

```
FILENAME: CalculationTests

FLOW NAME: Somar 3 + 3 $<
    EXECUTE: Click Button 3
    EXECUTE: Click Button Plus
    EXECUTE: Click Button 3
    EXECUTE: Click Button Equal
    EXECUTE: Verify Display 6
>$
```

Figure 4. Teasy – flow file

```
REGISTER PAGES:

    PAGE TO REGISTER: CalculatorHome
```

Figure 5. Teasy – *PageRegister*

```
CONFIGURATIONS

  INSERT BROWSER TO RUN TESTS: chrome
  INSERT MAX TIME TO WAIT ELEMENT (SECONDS): 5
  MAX AMOUNT TEST TO GENERATE: 300
  URL TO ROOT PAGE: http://localhost:3000/

  BROWSER CUSTOMER WIDTH: 1080
  BROWSER CUSTOMER HEIGHT: 720
```

Figure 6. Teasy – Configuration file

4. On using Teasy: a proof-of-concept

In this section, we present a *Proof-of-Concept* (PoC) related to Teasy use in real projects. The focus of the PoC is to analyze whether the Teasy Language can find errors in web applications, for this, a public project with injected mutants of different categories was used.

The PoC was performed by only one researcher and its main focus is to validate some benefits of using Teasy. Exploring mutation test concepts and a web application present in a public repository [1]. In addition, The structure[2] for use generated test scripts and Teasy[3] implementation also are in public repositories on GitHub.

4.1. Research Questions

This study addresses to answer the following *Research Questions* (RQ):

- **RQ01**: Is it possible to automate test scenarios using Teasy?
- **RQ02**: Is it possible to identify functional seeded-faults in real-world Web applications using Teasy?

4.2. Research Strategy

To find elements to answer (partially or completely) the aforementioned RQs, we followed a sixfold strategy:

- **Step 1:** search a Web Application in public code repositories;
- **Step 2:** collecting some software metrics from the web application selected: LoC (*Lines of Code*), technology, last commit date, amount of the classes, methods, and components;
- **Step 3:** To create testing scenarios using Teasy Language for the chosen application;
- **Step 4:** using a mutation testing strategy for Javascript to several other faulty-projects. Then, the "mutated" projects contain a modification that is expected to affect the original behavior of the project or even its visually aspect. To make this change, we use all the mutation operators present in [Mirshokraie et al. 2015]. For each method of the project, all the possible mutation operators were applied;
- **Step 5:** We run all generated test scripts on the original project and for each mutant made. For each project, the test scripts have generated a report with screenshots and step by step executed;
- **Step 6:** For last, we make an analysis in each report results and verifying manually in error occurrence to detect the reason and summarize this data.

4.3. Subject Application

In this PoC we have chosen a subject application developed using ReactJS [Fedosejev 2015] and available online in a public repository in GitHub. The application is a calculator and below one can find a set of the data/metrics extracted from it:

- **Application Name:** ReactJSCalculator[4];
- **Technology Used:** Javascript with ReactJS;
- **LoC (*Lines of Code*):** 13281;
- **Last Commit Date:** 15 Aug 2019;
- **Number of the classes:** 3;
- **Number of the Methods:** 7;
- **Number of the Components:** 3.

[1] See the Teasy evaluation project: https://github.com/yuryalencar/TeasyValidation

[2] See the structure for use in generated scripts: https://github.com/yuryalencar/TeasyStructure

[3] Link of the Teasy implementation: https://github.com/yuryalencar/Teasy

[4] Link of the project: https://github.com/yuryalencar/ReactJSCalculator

5. Study Results and Discussion

In this section we present a summary of the data extracted from the PoC. In addition to that, we present a discussion related to Teasy performance on finding faults in real applications.

5.1. Quantitative analysis of results

To present the quantitative result, we created the Table 1[5]. The table contains a component name and its respective modification to make the "Mutant project". The aforementioned table also contains the result of the executed test scripts generated by Teasy. The result was that the testing scenario we created from Teasy found **11 (eleven)** of the **14 (fourteen)** mutants created, which represents **78.57%** of accuracy regarding the given testing scenario for this web application.

Component	Mutant Operator	Result	Project Name
Calculator	Local/ Global Variable	Pass	Mutant1
Calculator	Local/ Global Variable	Pass	Mutant2
Display	Local/ Global Variable	Pass	Mutant3
Display	Dom	Fail	Mutant4
Button	Dom	Fail	Mutant5
Button	Local/ Global Variable	Fail	Mutant6
Button	Return Statement	Pass	Mutant7
Calculator	Conditional Statement	Pass	Mutant8
Calculator	Conditional Statement	Pass	Mutant9
Calculator	Local/ Global Variable	Pass	Mutant10
Calculator	Conditional Statement	Pass	Mutant11
Calculator	Local/ Global Variable	Pass	Mutant12
Calculator	Dom	Pass	Mutant13
Calculator	Local/ Global Variable	Pass	Mutant14

Table 1. Results of the run test scripts generated by Teasy

5.2. Subjective analysis of results and answers to RQs

Regarding the PoC's goal, the Teasy performance on detecting functional errors in web applications was satisfactory. The analysis of these results was simple to extract because the test scripts generated by Teasy create an HTML report with screenshots. Even though the scripts did not detect three of the mutants (project with some know seeded faults), the results are significant because mutants 4, 5, and 6, not present functional errors. Through a manual analysis of these case, we noticed these mutants changed the visual (appearance) of the application and the test scenarios were not created for detecting these problems.

Then, it is possible to mention that in this PoC, all of the possible-"tangible" faulty projects were detected through testing scenarios designed using Teasy. In addition to that, after performing all os the steps in Subsection 4.2 and analysing the special cases, we figured the following responses of the RQs: **RQ01**: the question has been answered completely: Teasy promote the creation of effective testing scenarios including adequate test cases; and **RQ02**: the question was answered completely and even the Teasy did not show 100% accuracy in the PoC's scenarios with the selected subject application, it is

[5]Link of the google sheets with a completes analysis: https://shorturl.at/evBNW

possible to identify functional failures. Regarding the undetected mutants, we noticed that they did not have any functional failures.

5.3. Threats to validity

Below we present the threats of this study on four different perspectives:

1. *Internal validity:* the PoC designed was exploratory and it focused on well defined research questions. The study provide evidences to answer the research question, then, we consider our internal validity high. 2. *External validity:* our study evaluate the effectiveness of Teasy regarding usability and effectiveness to apply MBT in real–world applications. However we have used only a single small web application, then we can claim moderate external validity. Further work is required including larger applications and considering other testing aspects. 3. *Construct validity:* once the concept of MBT, associated to mutation test is general to all applications, our construct validity is high. 4. *Conclusion validity:* our study presented a well-defined methodology and a step-by-step validation. All of the code generated and the screenshot are online e available. The, we can claim high conclusion validity.

6. Related Work

Among the related studies, we can highlight the following languages: *(i)* PARADIGM [Nabuco and Paiva 2014]; *(ii)* MIDAS [Herbold et al. 2015]; and *(iii)* WebSpec [Luna et al. 2011]. Considering a complete analysis regarding test data execution, report generation, user interaction modeling, among others, these languages are similar to Teasy. However, these languages are graphical, presenting the creation of automated testing scenarios using only graphic elements. Teasys advantage is due that textual languages are more productive in the test development scenario [Törsel 2013],

Among the existing textual languages, Gherkin [Wynne et al. 2017] and Legend [King et al. 2014] are the state-of-the-art. However, these languages do not have optimizations or abstraction of the test code. Thus, the tester still needs knowledge related to a programming language that will be used for automation.

7. Final Remarks

Due to the manual implementation and, mainly, the maintenance of automated tests, Teasy was proposed as an alternative to reduce the complexity and effort needed to perform automated functional tests for web applications. Through the PoC performed, we could verify its effectiveness in finding functional errors in a real application, based on these results we have evidence that the language can reduce the time in creating tests, but for a more assertive conclusion a greater number of subjects performing the PoC. In addition, Teasy's testing process allows for reporting with screenshots that make it easy to detect defective designs. As Teasy was designed using concepts related to the MBT approach, in future efforts we will use the pages and actions inserted in Teasy to perform an automatic sequence generation. Based on the expected results, we believe that the maturation of the Teasy project will allow the technological transfer from academia to industry.

References

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103.

Bisht, S. (2013). *Robot framework test automation*. Packt Publishing Ltd.

Delamaro, M., Jino, M., and Maldonado, J. (2013). *Introdução ao teste de software*. Elsevier Brasil.

Fedosejev, A. (2015). *React. js essentials*. Packt Publishing Ltd.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Herbold, S., De Francesco, A., Grabowski, J., Harms, P., Hillah, L. M., Kordon, F., Maesano, A.-P., Maesano, L., Di Napoli, C., De Rosa, F., et al. (2015). The midas cloud platform for testing soa applications. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8. IEEE.

King, T. M., Nunez, G., Santiago, D., Cando, A., and Mack, C. (2014). Legend: an agile dsl toolset for web acceptance testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 409–412. ACM.

Luna, E. R., Rossi, G., and Garrigós, I. (2011). Webspec: a visual language for specifying interaction and navigation requirements in web applications. *Requirements Engineering*, 16(4):297.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specifi c languages. *ACM computing surveys (CSUR)*, 37(4):316–344.

Mirshokraie, S., Mesbah, A., and Pattabiraman, K. (2015). Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering*, 41(05):429–444.

Nabuco, M. and Paiva, A. C. (2014). Model-based test case generation for web applications. In *International Conference on Computational Science and Its Applications*, pages 248–262. Springer.

Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill higher education. McGraw-Hill Education.

Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition.

Törsel, A.-M. (2013). A testing tool for web applications using a domain-specific modelling language and the nusmv model checker. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 383–390. IEEE.

Utting, M. and Legeard, B. (2010). *Practical model-based testing: a tools approach*. Elsevier.

Vani, B., Deepalakshmi, R., and Suriya, S. (2013). Web based testing — an optimal solution to handle peak load. In *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, pages 5–10.

Wynne, M., Hellesoy, A., and Tooke, S. (2017). *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf.

Živanov, Ž., Rakić, P., and Hajduković, M. (2008). Using code generation approach in developing kiosk applications. *Computer Science and Information Systems*, 5(1):41–59.