FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

From Requirements Specification to Tests Execution: An Integrated Approach from Acceptance Tests Perspective

João Gabriel Longo de Miranda

Mestrado em Engenharia de Software

Supervisor: Professor Ana Cristina Ramada Paiva

Second Supervisor: Professor Alberto Manuel Rodrigues da Silva

October 25, 2020

From Requirements Specification to Tests Execution: An Integrated Approach from Acceptance Tests Perspective

João Gabriel Longo de Miranda

Mestrado em Engenharia de Software

October 25, 2020

Abstract

Software requirements engineers and testers often define technical documents in natural language. However, this can cause problems, such as ambiguity, incompleteness, and inconsistency in the documentation, which, in turn, can lead to implementation errors. Also, to define acceptance tests, testers need to read and interpret these documents, and if they are not clear enough, test cases will also have the same problems, and the final system will be different from the intended one.

Previous research has shown that writing requirements and tests in a structured way, with controlled natural languages like ITLingo, can help mitigate these problems. Once all of the requirements are at the end of their design, they can be validated by the RSL language, which allows identifying and correct inconsistencies. It has also been demonstrated that it is possible to define test cases directly in these documents, and once these tests are written, the test scripts are automatically generated.

This study seeks to discuss new experiments to validate the language RSL with its complementary tools (called "ITLingo Studio") to be applied in different systems and technologies. Furthermore, using the preliminary results as indicators and motivators of new optimizations to the tool improves its current capabilities. Also, new features are presented in the program.

A different line of thought is reviewed related to applying test automation capabilities to security tests. This approach is later validated in a fictitious virtual store, namely the Juice Shop application, which serves as a test system and provides security flaws that allow the verification of this new approach.

In the end, it was possible to validate that the optimizations related to the full automation of some tasks and the implementation of new tools to support the execution of the tasks proposed by ITLingo. This represents a significant decrease in the time that users need to perform all of the ITLingo Studio steps.

Keywords: RSL, Tests Automation, Tests, Requirements Engineering, Acceptance Tests, Requirements to Tests

Resumo

Engenheiros de requisitos e testadores de software geralmente definem documentos técnicos em linguagem natural. No entanto, isso pode causar problemas, como ambiguidade, incompletude e inconsistência na documentação, o que, pode levar a erros de implementação. Além disso, para definir os testes de aceitação, os testadores precisam ler e interpretar esses documentos e, se não forem claros o suficiente, os casos de teste também terão os mesmos problemas, e o sistema final diferirá do pretendido.

Pesquisas anteriores mostraram que escrever requisitos e testes de forma estruturada, com linguagens naturais controladas como ITLingo, pode ajudar a mitigar esses problemas. Uma vez que todos os requisitos estejam no final do seu desenvolvimento, eles podem ser validados pela linguagem RSL, o que permite identificar e corrigir inconsistências. Também foi demonstrado que é possível definir casos de teste diretamente nesses documentos e, uma vez que esses testes são escritos, os scripts de teste são gerados automaticamente.

Este estudo busca ir além, discutindo novas experiências realizados para validar a linguagem RSL com as suas ferramentas complementares, (denominadas "ITLingo Studio") podem ser aplicados em diferentes sistemas e tecnologias, e utilizando os resultados preliminares como indicadores e motivadores de novas otimizações da ferramenta, visando aprimorar as suas capacidades atuais. Além disso, novas funcionalidades são apresentadas no programa.

Uma nova linha de pensamento é avaliada em relação à aplicação de recursos de automação de teste aos testes de segurança. Esta abordagem é posteriormente validada numa loja virtual fictícia, nomeadamente a aplicação Juice Shop, que para além de servir como sistema de teste, apresenta falhas de segurança que permitem a verificação desta nova abordagem.

Ao final, foi possível validar que as otimizações relacionadas à automação completa de alguma tarefa e a implementação de novas ferramentas para apoiar a execução das tarefas propostas pelo ITLingo realizadas representam uma diminuição significativa no tempo que os utilizadores precisam para realizar todas as das etapas do ITLingo Studio.

Keywords: RSL, Automação de testes, Testes, Engenharia de Requisitos, Testes de Aceitação, Requisitos para Testes

Acknowledgements

At the end of this academic journey, I would like to thank Professor Ana Cristina Ramada Paiva for her vision and wisdom throughout this master's degree, and all the other Professors who, in their area of expertise, brought new knowledge and opinions. I would also like to thank Professor Alberto Manuel Rodrigues da Silva, who, despite not being a professor at FEUP, was always available and committed to the project.

I would like to thank my parents, who have always supported me unconditionally in my choices, even when choosing to take their master's degree. I thank my sister that even far away, I know that she will always be supporting me. I also want to thank all my family members, who are still there for me.

I would also like to thank my fiancée that, with her unconditional support, helped me get here and encouraged me to keep going forward in the most challenging moments of this Masters' Degree and, ultimately, this research.

João Miranda

"We are what we repeatedly do.
Excellence, then, is not an act, but a habit."

Will Durant

Contents

List of Figures

List of Tables

Listings

List of Charts

Abbreviations

RSL	Requirements Specification Language
OJS	OWASP - Juice Shop
RE	Requirements Engineering
SRS	Software Requirements Specification
NLP	Natural Language Processing
SUT	System Under Testing

Chapter 1

Introduction

This research addresses the Software Testing and Requirements Engineering domains. This chapter overviews of the work done by presenting its context, problems addressed, motivation, and objectives.

1.1 Context

The vast majority of requirement specifications are written in natural language. However, the use of natural language may lead to problems, such as ambiguity, incompleteness, inconsistency, and incorrectness [10] in the documentation, which, in turn, can lead to implementation errors. Also, to define acceptance tests, testers rely on reading these documents. If the documents are not clear and complete, the test cases will also experience the same problems, and the final system will be different from what is intended.

One of the approaches to mitigate these issues is by using controlled natural languages, like the RSL (Requirements Specification Language) [6, 8], which provides a controlled way of writing both requirements and test cases [8], and recently was extended to allow the creation of test case specifications directly through it. RSL has a varied set of features and constructs that can be arranged into views that exist at different levels of abstraction.

RSL developed in the scope of the ITLingo initiative that has, at its core, the research, development, and application of specification languages in the IT domain, specifically in Requirements Engineering, Test Engineering, and Project Management [6, 8]. It uses an approach that improves technical documentation and promotes productivity and quality through re-usability and semi-automatic techniques.

At the end of the requirements and test specification phase through RSL is the software testing phase, which can evaluate the software development process by measuring success and failure rates through continuous testing. This action helps to maintain product quality as it can alert the team of defects early.

The goal of maintaining quality through constant testing can quickly become a time-consuming activity as systems evolve and become more complex. This scenario, coupled with the constant introduction of new requirements engineering artifacts, can quickly overwhelm the people in charge of the quality of a project, and because modern applications become more and more complete and complex, the best solution to solve such an issue would be to the possibility to carry out automation at least in the specification of tests and at most the generation of such test cases.

RSL makes possible to write the requirements of a system in a structured and controlled way and, at the end of this process, writing of test scripts based on steps that, in the end, will be transformed into real automation scripts (ROBOT Framework) and consequently allowing a faster automatic execution of tests.

1.2 Motivation and Objectives

Recently, Maciel et al. discussed an approach that aims to integrate and combine requirements and test cases specification with test automation tools, to improve the process of defining and executing acceptance tests [24, 32]. In particular, they discussed how to integrate ITLingo Studio with the Robot Framework [6].

However, this approach was straightforward and focused on the ability of ITLingo Studio, and consequently, RSL, to automatically generate and execute test cases. In other words, to run an end-to-end RSL documentation plus test scenario running, it is necessary to run the language as a secondary Eclipse application, which is used to develop the use cases and test cases. After the requirements are correctly described and mapped, a new file is generated containing the Robot Framework script. Then a third party chrome extension called *Web Scrapper* [44] is used. This extension can create CSS mappings of a website and then export the mappings generated to JSON format. This extension brings up the first issue. On modern applications such as SPA based systems (Angular, React, VueJS), CSS mappings may not be the best choice as many times the ids may be generated dynamically.

Further on, after the mappings are made, two scripts must be executed; the first one will transforms the JSON file into a XML file. Furthermore the second one will read the generated XML and map insert then on the generated Robot Framework file based on the keyword selectors.

This part of the process brings an unnecessary extra step: the mapping from JSON to XML when the JSON could be directly mapped into the Robot Framework file.

After all of that is done, it is necessary to run the robot framework through the terminal.

Thus, this work aims to assess the applicability of the approach in different settings and applications that can be considered more recent by applying it to an application developed in Angular.

Angular is a front-end framework developed by Google and takes advantage of the most recent web technologies.

That being said, the improvements that will be discussed in more details through this dissertation can be mapped to three main areas:

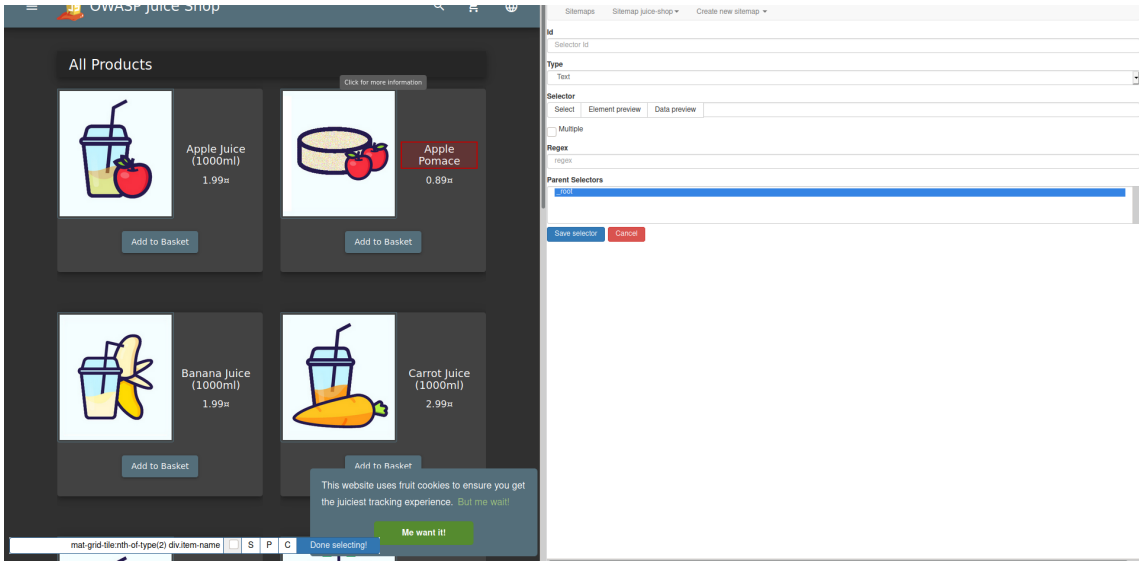

Figure 1.1: Visualization of the webscraper extension workflow [44].

- The complete automation of tasks that right now are repetitive and require manual input (such as JSON generating, JSON mapping to the ROBOT file) improve the productivity of the tool.

- The creation of a web browser extension that is capable of extracting element data directly from web applications and is designed to be closely integrated to the ITLingo Studio

- The implementation of a Web IDE that can take leverage of all of the features already implemented to ITLingo Studio and RSL, and improving the accessibility of the tool, through the possibility of being hosted and accessible.

The other line of discussion on this dissertation is a preliminary check that aims to validate if it is possible to extend the current testing automation approach to perform security testing that, if possible, could introduce an entirely new line of research and allow ITLingo Studio and RSL to have even more capabilities, improving its ability to provide quality to software projects [27].

1.3 Document Structure

This dissertation is structured in 5 chapters.

Chapter 2 presents the current state of the ITLingo Studio and its RSL language.

Chapter 3 overviews of other related works and tools that somehow contribute to the results of this research.

Chapter 4 presents the modifications made to the ITLingo Studio tool and the new features added to it.

Chapter 5 describes experiments that are performed using the new improvements and the older ITLingo Studio for comparison while trying to answer three main questions raised by this dissertation.

Chapter 6 concludes this dissertation and looks forward to what is ahead to the ITLingo Studio tool.

Chapter 2

Background

The proposed testing process intends to combine (1) both requirement and test specifications, defined in a tool like ITLingo Studio, with (2) test automation, supported by a tool like Robot Framework. This section briefly introduces the tools considered in this research, and overviews the proposed process.

2.1 ITLingo Studio and Requirements Specification Language

ITLingo-Studio is an Eclipse-based tool (i.e. a desktop IDE) for authoring IT technical documentation, such as requirements and tests (with the RSL language), project management plans, or platform-independent application specifications (with the ASL). We only consider the ITLingo RSL (or just RSL for brevity) in this research's scope. RSL is a controlled natural language that helps the production of requirements and tests specifications in a systematic, rigorous and consistent way [10, 6, 8, 7]. RSL includes a rich set of constructs like use cases, goals, user stories, but also use case tests, data entities, actors, stakeholders, and many others (further details in [36, 31]. The ITLingo languages have been implemented with the Xtext framework (https://eclipse.org/Xtext/), so its specifications are rigorous and automatically validated and transformed into other representations and formats. Figure 2.2 depicts the ITLingo Studio with a test case specified in both RSL and Robot Framework languages.

RSL, by definition, is a research initiative that intends to develop and apply rigorous specification languages for the IT domain, such as requirements and test engineering, with RSL [24] and one of its main advantages is that it helps standardize the documentation process while improving business knowledge of requirements engineers. Another critical point is that it also allows for easy documentation learning and maintenance, as any requirements engineer would be familiar with the form of writing (which most often does not happen in natural language documents).

The effort needed to produce formal requirements specifications document is substantial. Also, many of the Requirements Engineering tasks are manual on nature, and when a large amount of information is involved, this could cause catastrophes in projects. Thus, the RSLingo approach proposed that some manual tasks be automated with the following purposes [10]:

```
Package WebApp


/*******************************************************
   System definition
*******************************************************/

System MyStoreWebApp "Juice Shop (Application Level)" : Application [ isFinal
    description "Juice Shop is a fictitious web store for penetration testing
                 testing purposes that can be used to automation testing approaches too"]


/*******************************************************
   DataEntities view
*******************************************************/

DataEntity e_User "User" : Master [
    attribute ID "ID" : Integer [isNotNull isUnique]
    attribute email "Email" : Regex [isNotNull isUnique]
    attribute password "Password" : Regex [isNotNull isEncrypted]
    attribute securityQuestion "SecurityQuestion" : Text [isNotNull]
    attribute securityAnswer "SecurityAnswer" : Text [isNotNull]
    primaryKey (ID)
    description "Users"
]

DataEntity e_Address "Address" : Master [
    attribute ID "ID": Integer [isNotNull isUnique]
    attribute country "Country": String [isNotNull]
    attribute recipientName "RecipientName": String [isNotNull]
    attribute mobileNumber "MobileNumber": String [isNotNull]
    attribute zipCode "ZipCode": String [isNotNull]
    attribute address "Address": String [isNotNull]
    attribute city "City": String [isNotNull]
    attribute recipientState "RecipientState": String [isNotNull]
    primaryKey (ID)
    description "Addresses"
]

DataEntity e_PaymentDetail "PaymentDetail": Master [
    attribute ID "ID": Integer [isNotNull isUnique]
    attribute holderName "HolderName": String [isNotNull]
    attribute cardNumber "CardNumber": String [isNotNull]
    attribute expiryMonth "ExpiryMonth": Integer [isNotNull]
    attribute expiryYear "ExpiryYear": Integer [isNotNull]
    primaryKey (ID)
    description "Payment Details"
]
```

Figure 2.1: RSL specification.

```
*** Settings ***
Documentation               This is a basic test
Library                     Selenium2Library
Variables                   JuiceShop.py

*** Test Cases ***


SignUp-Test_1
    [Documentation]         As a Visitor I want to sign up
    open browser    https://juice-shop.herokuapp.com/#/register  ${Browser}
    Sleep  1s
    Click element  ${t_uc_1_Dismiss}
    Sleep  1s
    input text   ${t_uc_1_email}   ${t_uc_1_email1}
    Sleep  1s
    input text   ${t_uc_1_password}   ${t_uc_1_password1}
    Sleep  1s
    input text   ${password_confirmation}   ${password_confirmation1}
    Sleep  1s
    Click element  ${SecurityQuestion}
    Sleep  1s
    Click element  ${SecurityAnswer}
    Sleep  1s
    input text   ${t_uc_1_answer}   ${t_uc_1_answer1}
    Sleep  1s
    Click element  ${t_uc_1_Confirm}
    Sleep  1s
    Page Should Contain Element ${SignUp_PointAndClick}  ${success_message1}


SignIn-Test_1
    [Documentation]         As a Visitor I want to sign in
    open browser    https://juice-shop.herokuapp.com/#/login  ${Browser}
    Sleep  1s
    Click element  ${t_uc_2_Dismiss}
    Sleep  1s
    input text   ${t_uc_2_email}   ${t_uc_2_email1}
    Sleep  1s
    input text   ${t_uc_2_password}   ${t_uc_2_password1}
    Sleep  1s
    Click element  ${Confirm_SignIn}
    Sleep  1s
    Click element  ${My_Account}
    Sleep  1s
    Page Should Contain Element ${SignIn_PointAndClick}  ${t_uc_2_accountEmail1}

```

Figure 2.2: Test script in Robot.

- **Domain analysis:** in order to identify all of the relevant concepts, their relations, and how the software manipulates theme to provide the desired capabilities while respecting all of the established constraints regarding the functionality that provides those capabilities to the users [10];
- **Verification:** in order to perform consistency checks on the extracted domain knowledge, through inference and ambiguity resolution based on glossaries and general lexical resources, to prevent delays and extra costs caused bu discovered defects [10];
- **Transformations:** in order to automatically generate alternatives to the requirements representations such as diagrams, tables, reports, and others in a controlled natural language, according to specific viewpoints [10];

Developed using Xtext, which is a tool that helps create new language syntax, RSL can be used to reverse engineer a system's documentation through legacy databases; generate new requirements from spreadsheets templates; once written, this documentation can be exported to natural language templates; and write new documentation through this strict language, which we will be exploring further on. It is a tool that proposed to use simplified natural language processing and human-driven techniques to capture relevant information from ad-hoc natural language requirements specifications [8].

This technique is possible by using two original languages: RSL-PL (*Pattern Language*, used for encoding requirements engineering specific linguistic patterns, and RSL-IL (Intermediate Language), created to address requirements engineering concerns [8]. Through these two languages and the mapping created between them, it is possible to extract, analyze and convert requirements initially written in natural language into a structured format, thus reducing ambiguity and creating a more thrust worthy SRS.

A new language was created through these languages, based on the RSLingo initiative and was called RSL. This new language aims to help the production of SRSs in a systematic, rigorous, and consistent way [8].

RSL can also generate scripts based on the Test Specification Language (TSL) and Robot Framework.

RSL supports the specification of the following business-related and system-related concerns:

RSL has its own way of defining Data Entities (*DataEntity*) [8], Use cases (*UseCase*), and State Machines (*StateMachines*).

2.2 From Requirements to Tests: Previous Approach

Figure 2.4 shows the approach originally proposed in [24], which considers an "end-to-end integration of requirements, test cases, and test scripts", namely by combining tools like the ITLingo Studio (with RSL specifications) and Robot Framework (with keyword-based test scripts).

This approach consists of a sequence of tasks that vary from the specification of requirements until the execution of tests.

Concerns / Levels	Package	Context	Active Structure (Subjects)	Behavior (Verbs, Actions)	Passive Structure (Objects)	Requirements
Business	package-business	Business SystemRelation BusinessElement Relation	Stakeholder	BusinessProcess (BusinessEvent, BusinessFlows)	GlossaryTerm	BusinessGoal
System	package-system	System Requirement Relation	Actor	StateMachine (State, Transition, Action)	DataEntity DataEntityView	SystemGoal QR Constraint FR UseCase UserStory

Figure 2.3: RSL Levels and Concerns [8, 9].

Requirements Specification. In the task "(1) Specify Requirements (manual)" the requirements are elicited and documented by requirement engineers using the RSL language. It may also involve other stakeholders (like testers or domain experts) for validation purposes. This task favors a systematic specification of requirements and can be done manually with the ITLingo Studio (an Eclipse IDE for authoring RSL specs) or an Excel template.

Test cases specification and validation. Task "(2) Specify Test Cases (semi-auto)" uses the requirements defined in the previous step to generate test cases, namely considering the mappings between use case and use case test, as defined in [8]. Task "(3) Refine Test Cases (manual)" allows the tests engineers to refine and complete the (previously generated) test cases with all the relevant test scenarios, steps, etc. Also, the tests generated from the previous task may be subject to manual validation and this could result in new test cases which are then added to the final specification. This test case validation is also a human-intensive and time-consuming task. *Test scripts generation*. Once the specification and validation of test cases are completed, the task "(4) Generate Test Scripts (auto)" consists in the generation of test scripts, which can be then executed by a test automation tool, like the Robot Framework. This generation is based on the mappings established between the RSL specification and the Robot Framework's concrete syntax [24].

Map the UI elements. The user interface (UI) of a web application involves several UI elements [24]. To properly support the test automation based on the acceptance tests, these generic UI elements have to be mapped to concrete UI variables defined in the test scripts to act over related UI elements during test case execution. The establishment of this map is the purpose of task "(5) Map GUI elements to keywords (semi-auto)". However, using a web scrapper browser extension, as the Selenium WebDriver (https://www.selenium.dev/projects/), it is possible to automatically capture and generate scripts that successfully map these UI elements with the test scripts. Unfortunately, most of these cases depend on the proper keywords mapping, which is hard to be fully automated.

Execute test scripts. Task "(6) Execute tests (auto)" takes into consideration the test scripts (previously generated, and with their UI variables mapped to concrete UI elements) and execute them on the system under test (SUT). This test automation task produces an artifact "Test Report",

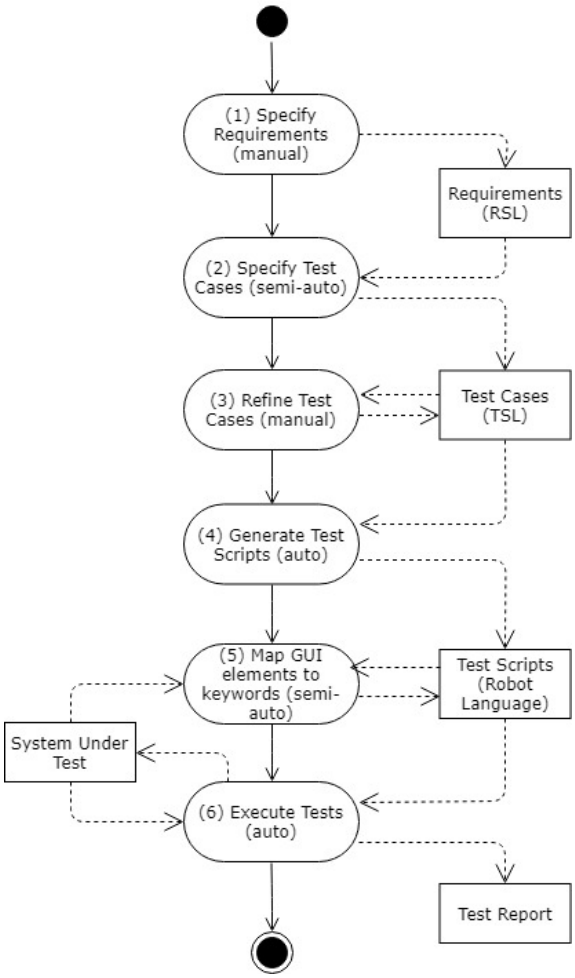

Figure 2.4: Proposed testing approach, as originally defined in [24].

which includes the log details of its execution.

Chapter 3

State of The Art

This chapter introduces the state of the art research and reviews the existing literature, and then analyses some similar approaches to the current scenario of the ITLingo Studio, in an attempt to create correlations that can later be applied in this scenario. Finally, it presents some tools that can be used to keyword mapping and set the baseline to the implementations made further on to the ITLingo Studio needs and requirements.

3.1 Test Cases generation from Requirements

In a typical project scenario, once a Requirements Engineer finished documenting a specific part of a system, which usually is achieved through natural language specification, a Tester is required to read this document in order to validate it and after it is formally approved, derive the test cases from the document.

Writing requirements through natural language is a prevalent practice worldwide, but this approach, although common, is error-prone and may cause many issues, such as lack of standardization on documents, ambiguity, incompleteness, inconsistency, and incorrectness [10]. Ambiguity is particularly a severe problem because it allows multiple interpretations of the same requirement [10], which leads to testing scenarios that are inconsistent with the result of the development.

Below are presented some strategies that can be used to generate test cases from requirements. There are other strategies that are also worth mentioning, such as the ones that generate test cases from models [28, 29, 4] or from execution traces [32, 1, 37].

3.1.1 Test Cases from Natural Language Specification

In the subject of natural language and its standardization and ambiguity issues, Ansari et al. discussed in [3] that if a certain level of standardization in requirements engineering is achieved, and by developing a Natural Language Processing tool which is capable of analyzing such documents and extract data from this assets based on keywords, it could possible to generate test cases directly from the requirements [3].

This approach shows itself more feasible because requirements are typically written in a non-standardized way by many different Requirement Engineers, and it makes the automation task more difficult to be accomplished if full textual and contextual processing [3]. Such an analysis would be more complicated than the NLP and involve developing an Artificial Intelligence capable of understanding and processing what was written by different engineers.

As mentioned, Ansari et al. proposed that by extracting keywords from the requirements, the test scenario generation time-need, effort, and cost could be considerably reduced [3]. However, the NLP still expects a functional requirement document written to a certain level of standardization in order to be able to read and analyze the document correctly. Otherwise, no test cases could be output [3].

In general this approach is done in three steps:

1. Functional Requirements are written and provided as input to the NLP;
2. NLP analyzes the requirements, and extract the key data if possible;
3. the test case data is built and output based from the extracted key data from the requirements;

This study expects that the requirements are written in a particular standardized form so that the AI algorithm can recognize the semantics and the specification context. In other words, this could allow automating test execution, but not all of the patterns, and still, it would need a certain degree of standardization in order to accomplish something close to full automation.

That being said, this could be applied to the natural language part of the RSL tool to standardize better and validate the completeness of the requirements and test cases that were written and generated, which could give a good leap of quality in the overall result.

3.1.2 Test Case Generation from High Fidelity Prototypes

As proposed by [38], requirements are extracted from prototype requirements, which are then converted into a canonical model known as Abstract Screen Model (ASM) [38]. Abstract Screen Models are elements that are able to capture navigation events from a program in order to be used to generate Screen Navigation Models which is the structure that defines the flow between two screens [38].

As depicted in **Figure 3.1**, ASMs are used to capture navigation event information that, in turn, will be used to generate the Screen Navigation Model, which depicts the navigational flow between screens. The ASMs also capture information related to the flow behavior between UICs on a screen [38].

Then on another step, a tool called Screen Model Generator will be used the information previously created to generate a flow graph. Once the Screen Model for all screens in a prototype is generated, they can be combined to generate a holistic flow graph called Application Model (AM) [38]. This application model is then traversed by the Graph Traverser toll, which derives test paths. Finally, the Test Case Generator Tool can transform those test paths into human-readable test cases.

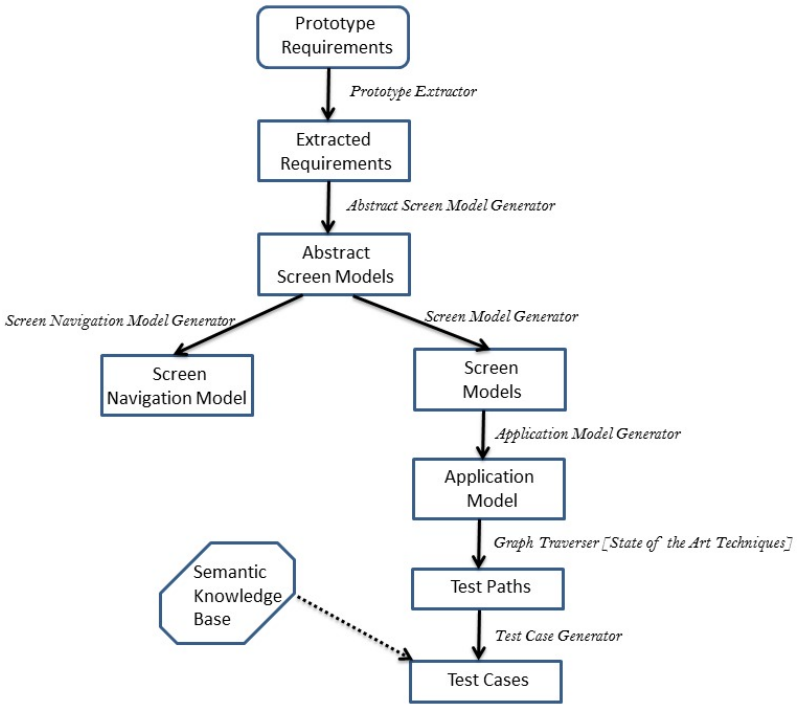

Figure 3.1: Proposed prototype to Test-Cases Generation Process. [38]

In the end, this tool is capable of automating the test case generation, but the final result still needs to be manually executed by Testing Engineers, this in part, could reduce the effort expected from testers to accomplish their tasks. However, this method could also be expanded to generate automated test scripts based on the high fidelity functional prototype.

3.1.3 Use-Case driven test generation

In [30] it is proposed a method using Use cases to generate tests that works in phases.

The first part of the proposed method ((a) to (c)) aims at generating test objectives from a use case view of a system. The use case diagram should be a global view describing the system's primary functions [30]. It is then proposed a requirement-by-contract approach. These contracts should not be executed before and after the execution of a use case, but are used to infer the correct partial ordering of functionalities that the system should offer [30].

From the use cases and their contracts, a prototype tool builds a simulation model (b) and generates correct sequences of use cases (c) [30]. Such a sequence is called a *test objective*. This phase should allow the requirements engineer to check and, if necessary, fix the requirements before the tests are generated from them [30].

In the second part (from (c) to (e)), the test scenarios are generated. Here, the test scenarios may be used as an executable test case directly, or it may need some additional information from the tester, i.e., if additional parameters are required [30]. In order to get scenarios from the objectives, more information is needed. This information can be provided through sequence diagrams, state machines, or activity diagrams. All of those artifacts describe the scenarios that correspond

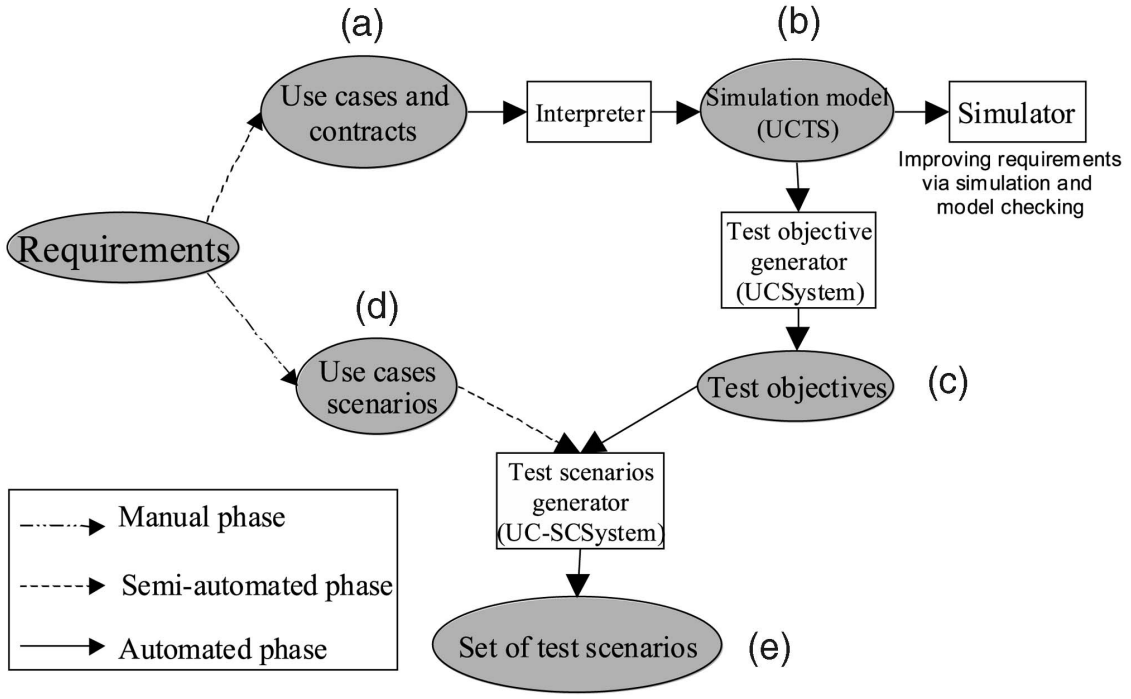

Figure 3.2: Proposed methodology by [30]

to a use case [30]. The transformation from objectives to scenarios consists of replacing each use case of the objective by one of its use case scenarios, using a tool proposed in [30].

This method results in system test scenarios with embedded oracle function [30], and several points had to be taken into account:

- *Use case and contract validation:* The use cases can suffer validation through simulation and model-checking. The underlying model has to be compact enough to avoid the combinatorial explosion of the simulation model's internal states, called UCTS (Use Case Transition System). This point is overcome by the two-step approach, which divides the complexity of high-level and detailed requirements into two levels (use cases and sequence diagrams), and by the introduction of use, case parameters to deal with central systems concepts and actors [30].

- *Definition of system test criteria:* Based on the UCTS model, test generation criteria has been proposed that automate the production of test objectives. The most efficient criteria have been identified through experimental comparisons [30].

A use case mainly depends on actors related to it and the business level concepts it is required to handle; therefore, actors involved in a use case can be considered parameters [30]. After these parameters are specified, contracts are created in the form of preconditions and postconditions involving the parameters. These contracts are logical expressions combining predicates with logical operators [30]. In this case, Boolean logic is used, so the predicate is either true or false (never

undefined) [30]. The preconditions guard the use case execution while the postcondition specifies new predicate values after execution.

Although this methodology does not generate the tests themselves, it should be possible to derive them from the automated generation of test cases described above.

3.1.4 Behavior Driven Development

BDD, which has been developed from the test-driven development technique, utilizes principles of user stories and the test-driven development approach as user stories technique is the foundation stone for the BDD testing scenario template, which is observable from a comparison of BDD and user story template [26].

Once the scenarios are completed, they can be expressed in a format that is readable by non-technical employees, and still is in such a template that it is executable by BDD tools such as Cucumber. This format, called the Gherkin syntax, can serve multiple purposes at once [2]:

1. The scenarios act as executable specifications for the behavior of the feature under test [2].
2. These specifications can be executed as automated regression tests [2].
3. The scenarios act as documentation about the feature that follows the main code in a version control system [2].

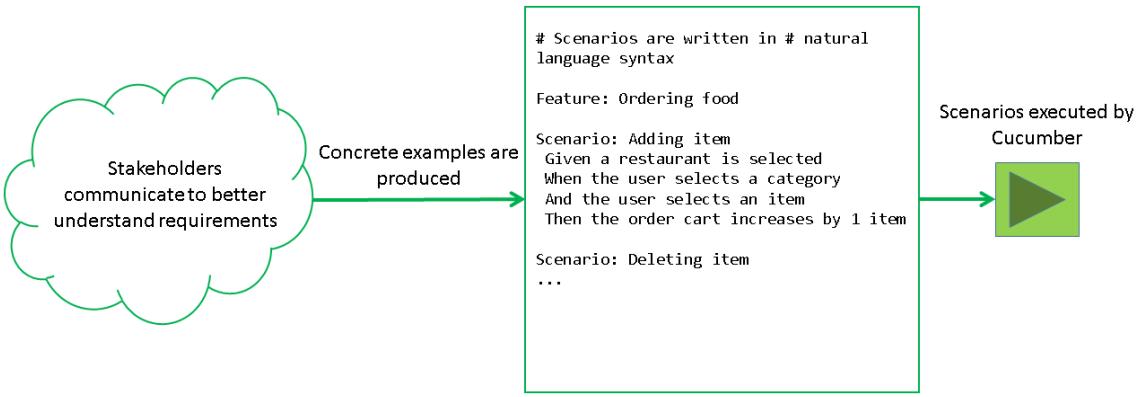

Figure 3.3: A BDD workflow [2]

The steps as defined in **Figure 3.3** specify what the scenario is, what assumptions it uses, and how the feature will behave in terms of the outcome [2].

By default, BDD follows the syntax of domain-specific language Gherkin, and it is executable through Cucumber (although there are others). By generating acceptance test cases through the BDD documentation approach, it is possible to define the system under development as a black-box and test it against the desired functionalities.

Gherkin is a domain-specific language that helps a Requirements Engineer describe business behavior without going into details of the technical implementation. This text acts as documentation and template for the automated acceptance tests. Gherkin could be easy enough to be

implemented and understood by the tools, the developers, the requirements engineers, and the testers.

Two of Gherkin's main disadvantages are that it may not work well in all scenarios, and poorly written tests can quickly increase the test-maintenance cost by reworks.

In the case of the research in question, it should be noted that BDD would be a supporting feature further to enhance the toolset's testing and documentation capabilities.

3.1.5 Keyword Driven Testing

KDT aims at separating test design from technical implementation [34]. The goal here is to limit the exposure to unnecessary details and avoid duplication. It allows tests to be written easier, creates more maintainable tests, and enable experts from different fields and backgrounds to collaborate [34].

As mentioned in [34] keywords can be separated in 7 categories

Table 3.1: KDT Test Table

Label	Explanation
ACTION	Keyword performing an action on the System Under Testing (SUT) capable of modifying its state.
ASSERTION	Keyword verifying that a predicate is true at a specific point of test execution.
CONTROLFLOW	Keyword allowing to modify the control flow of the test execution.
GETTER	Keyword allowing to extract an element from the SUT.
LOGGING	Keyword dumping logs during execution
SYNC	Keyword relating to the synchronization between the SUT and the tests.
USER	Keyword created by a user.

[a] As in [34]

Focusing in the *GETTER* this keyword allows for the tool to extract the elements from the System Under Testing but still it requires manual input to correctly map the elements from the SUT to the acceptance test in order.

3.2 Tools

One of the challenges presented by this research was a form to optimize the keyword mapping process which is the process of introspecting into a website and generating keys that represent a single component in a view and using it to reach such element in another moment. This mapping can be done through different strategies, such as CSS mapping and XPath. Maciel et al. proposed in [24] the usage of CSS mappings as the main way of selecting the element in the UI, but this

```
*** Settings ***
Library           Selenium2Library

*** Test Cases ***
Valid Login
    Open browser to login page
    User "demo" logs in with password "mode"
    Welcome page should be open

*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}
    Login Page Should Be Open

Login Page Should Be Open
    Title Should Be    Login Page

Go To Login Page
    Go To    ${LOGIN URL}
    Login Page Should Be Open

User "${username}" logs in with password "${password}"
    ...

Welcome Page Should Be Open
    ...

*** Variables ***
    ${SERVER}         localhost:7272
    ${BROWSER}        Firefox
    ${DELAY}          0
    ${LOGIN URL}      http://${SERVER}/
    ...
```

Figure 3.4: Script based on the Keyword Driven Test Case Table [34]

usage created an issue that presents itself in the fact that CSS mappings may not be as reliable in complex scenarios as they were in legacy applications. This issue required the introduction of new tools in a preliminary phase that supported the conception of the ITLingo browser extension which is focused in keyword mapping.

3.2.1 Selenium

Selenium is a web testing tool that uses scripts to run tests directly within a browser. It is compatible with various Programming Languages such as C#, Java, Python, and others. By default, the Selenium driver uses JavaScript and iFrames to embed the test automation engine into the tested page. It allows the same test scripts to be used to test multiple browsers and platforms [21].

The library offers the developer a series of commands that can be used to manipulate and validate the current state of a web page to execute acceptance tests. There is now a Selenium Web IDE that can manually record steps and then automatically generate Selenium powered test cases.

Selenium tests are easy to write because it allows identifying DOM elements, so tests can be written using specific identifiers of the needed elements [21]. The tool can provide some automation level to infer elements from the screen without using IDs exclusively, but this approach can generate queries that negatively affect the overall testing scenario [21].

One of the main advantages of selenium testing scripts is that by being easy to write, in other words, a tester should be capable of writing a Selenium test without knowing what the implementation will be. However, this practice still raises some issues, such as the name of the fields, which

would not naturally be identified by the name that the tester chose, or the test command used was not adequate for the tested functionality.

Selenium, as a tool, is one of the main components that can benefit the research in the form of test automation. With its scripting tool, it should be possible to implement the test generation further and even expand the types of tests implemented by the tool proposed by [24].

Recently the developers behind Selenium released a new tool called Selenium IDE, an open-source capture and replay test automation tool for the web [16]. It is developed natively as a browser extension and can be used to create test scripts through recording. It is a testing automation tool that is simple and easy to use. Selenium IDE is based on the capture-and-replay technique and can be used for many kinds of testing [40, 16].

Selenium IDE is capable of generating CSS or XPath mapping locators, which provides a good set of options to run a test. These locators allow the Selenium IDE Automation tool to find the elements when its time to run the automated tests [13].

3.2.2 Webscraper

Web scraper is a browser extension that performs CSS mapping of screen elements and allows the exportation of such elements in a formatted JSON that later can be used to create a keyword/locator structure that can be used in other tools [44].

The extension takes leverage of a point and click interface which helps making data extraction from the web page as simple as possible. It helps building sitemaps from different types of selectors, and makes it possible to customize the data to be extracted to the limit. In the end, the data generated through the extension can be exported to JSON, CSV, and XLSX files [44].

3.2.3 Ruto XPath Finder

Ruto Xpath Finder is a browser extension that helps a user find unique locators and XPath that can be later used in test automation tools such as Selenium [41].

For testing automation, this extension is useful while writing test scripts as all a user needs to do is select the element that should be mapped and select one of the tool's locators. Ruto is also able to generate code snippets that can later be tested in other languages such as python or javascript, this can be a useful way to refine the exported data [41].

3.2.4 XPath Helper

XPath Helper is a Google Chrome extension that helps visualize elements through their XPath mappings [35]. It provides a UI that contains a text area where a query can be made, and if such query matches one or more elements on a screen, those elements get highlighted and listed. It is useful to validate and refine XPath queries that are automatically generated and not always are 100% accurate [35].

This extension could validate locators that are provided by the Ruto XPath Finder.

3.2.5 Robot Framework

Robot Framework (RF for short) is a generic open-source automation framework test automation and robotic process automation [33]. RF has an open and extensible architecture and can be integrated with other tools to create flexible automation solutions. Robot Framework provides a textual language with a keyword-based easy syntax, both machine and human-readable. Its capabilities can be extended by libraries implemented with Python or Java.

In the scope of our research, the test cases defined in RSL [36] are generated to RF test scripts, and then these test scripts are executed by the RF's engine [34]. A key advantage of RF is its high modularity and extensibility, as it is platform-agnostic and thanks to its driver plugin architecture, the core framework does not require any previous knowledge of the system under test [15].

3.3 Discussion

By combining some of the solutions presented above, such as [20] and the proposal to treat the models-driven testing at component level to get more flexible and reusable models, alternatively, [38] with his approach to generating the test cases based on functional prototypes to leverage the expected functionality and map it to its corresponding requirements and subsequent tests. Another proposal that is worth mentioning is [23] and their view on the mapping of screen elements to the test, which in turn could be combined with the approach offered by [34] which proposes keyword-driven test cases generation and [21], which proposes some other useful techniques that may prove useful along with their advantages and disadvantages.

It is possible to increase the level of independence of test case generation and validation algorithms [5, 25, 19]. However, it may not be possible to completely automate the mapping of GUI elements to test cases, as it is challenging to impose a standardization of element nomenclatures. It is also not a good practice to define UI element names in the documentation phase [28, 34].

Also, one of the essential points in this process would be the test validations view, with solutions as proposed in [5] and [25], it would be possible to transparently benefit test cases coverage while still allowing for Test Engineers to make manual validations when and where it is needed.

Chapter 4

ITLingo Tools Extensions

The first improvement in the process was to develop an extension that could read a previously generated JSON structure containing all the keywords in an array and insert them in a table which, in its turn, is used to map their corresponding X-Path.

The next improvements were the configuration of the Xtext to allow the generation of packaged .jar language servers, which allows other IDEs to execute and run the ITLingo Studio locally. Also, it was developed inside the 'RslGenerator.xtend' file new functions that allow for the automatic generation of JSON files containing the keywords used for mapping and a new function capable of identifying if there is a mapped JSON in a specific folder to replace the keywords with their mapping counterparts.

Finally, the last improvement involves implementing an Eclipse Theia extension that allows the IDE to execute and validate the written use cases and tests through the previously generated server. Then the code markup is defined that allows the IDE to perform code highlighting. The implementation of such extension should allow for many further implementations, such as diagrams generation, automatic changes listening to mapped items, and further other improvements to the workflow.

4.1 Changes to the ITLingo Studio Project

The Xtext technology, at its basic functionality, provides a compatibility layer that can automatically create an extension that is then executed through the Eclipse application to execute a Domain Specific Language (DSL). As Eclipse is primarily a fully-featured Java IDE, it may not be the best option to use a new DSL in a production environment. With that in mind, a few modifications were made to the current implementation of the ITLingo Studio RSL language to better support other technologies, such as Eclipse Theia.

The first step in the process was converting the RSL Xtext project to Xtext Gradle Project. Gradle is an open-source build automation tool [18] that, with the correct configurations, is able to compile the ITLingo Studio from source. Another advantage of this approach is that it allows for the program's packaging as an executable Language Server *'.jar'* and dependency resolver. Once

Gradle is integrated into the project, it is possible to build, package and execute tests all through its scripts.

After this step is concluded, a few modifications were made to the 'RslGenerator.rsl' file, which is responsible for generating all of the Robot framework scripts once a specification is completed. Two new functions were added, one that allowed the generator to create a new JSON file prepared to be copied into the extension previously described and used to map the keywords on a website. The other function was used to check if such a JSON file was placed in a folder and uses its data to create a new Robot framework file, but this time with the website's mapped data.

```
def jsonGenerator(Resource resource, IFileSystemAccess2 fsa) {
    val simpleClassName = resource.getURI.trimFileExtension.lastSegment

    for(useCaseTest : resource.allContents.filter(UseCaseTest).toIterable)
        for(scenario : useCaseTest.scenarios)
            for(step : scenario.testSteps) {
                if(step !== null && step.extension !== null && step.extension.target !== null && step.extension.target.variable !== null)
                    for(variable : step.extension.target.variable)
                        keys.add(variable.name)
                if(step !== null && step.extension !== null && step.extension.target !== null && step.extension.target.content !== null)
                    for(content : step.extension.target.content)
                        keys.add(content)
            }

    keys.add('Point&Click')

    fsa.generateFile(simpleClassName + '.json', '''
        {
            "selectors": [
                «FOR key : keys»
                {
                    "id": "«key»",
                    "selector": ""
                }
                «ENDFOR»
            ]
        }
    ''')
}
```

Figure 4.1: Function that is responsible for generating the JSON based on the extracted keywords from specification.

The idea behind this function is to iterate over all of the Use Case Tests, Scenarios, and Steps in order to extract all of the possible keywords based on their type. Later, they are added to a *HashSet*, which is a type of array that avoids repetitions and is finally printed as a JSON array and saved to a file.

The second function, as depicted in 4.2 is used to read the manipulated JSON, which now is saved in a folder called *src-run* and by iterating through its selector list members replaces the keywords on the original Robot Framework file and then creates a new one in the *src-run* folder.

These two steps together help to automate most of the process introduced in [24] as no scripts are needed to be manually run any longer.

4.2 ITLingo Studio browser extension

Chrome extensions can be considered compact software programs that can customize the browsing experience [17]. These can create new experiences or preferences in the browser and are built with standard web technologies, such as HTML, CSS, and JavaScript [17].

By definition, an extension should fulfill a single purpose and can include multiple components with many functionalities [17].

```
def robotMapper(IFileSystemAccess2 fsa, String simpleClassName) {
        var file = fsa.readTextFile('../src-run/' + simpleClassName + '.json')

        var robot = fsa.readTextFile('../src-gen/' + simpleClassName + '.robot').toString()
        var result = ""


        if(file != null && file.toString().length > 0) {
            var parser = new JsonParser()

            var jsonObj = parser.parse(file.toString()).asJsonObject

            var selectors = jsonObj.get('selectors').asJsonArray

            for(var i = 0; i < selectors.length; i++){
                var selectorObj = selectors.get(i).asJsonObject

                var id = selectorObj.get('id').asString
                var selector = selectorObj.get('selector').asString

                robot = robot.replace('[' + id + ']', selector)
            }

            fsa.generateFile('../src-run/r_' + simpleClassName + '.robot', '''
                «robot.toString()»
            ''')
        }
    }
```

Figure 4.2: Function that is responsible for reading the JSON which results from the mapping and returns the complete ROBOT file.

Chrome extensions can also be created to work on specific pages (through Page Actions), to run in the background (using Background Pages), and even modify an existing page loaded in the browser (using Content Scripts) [39].

The architecture of an extension depends on the required functionality and the most common components that an extension uses are manifest, background scripts, UI elements, and options page.

Manifest Every extension must have a JSON-formatted manifest file. This file provides essential information about the extension, such as the extension name, the permissions required to run it, the scripts that must be executed and injected, and which web pages the extension should be allowed to execute [17, 39].

Background Script This is the extension event handler. It contains many listeners used by the extension to communicate the scripts that are injected into the web page with the scripts that perform the actual functionality of the extension [17].

UI Elements Extension UI elements can be contained as popups or as DevTools pages, and they are used to execute the functions of the application [17]. Their lifecycle is isolated from the running web page.

Content Scripts These are the scripts that get injected into the web page and perform the UI Elements' needed functions. For example, in the ITlingo studio extension, an event handler is added to map page elements later relayed to the UI Elements through events and callbacks [17].

Options Page This page would allow configuring an extension [17]. It is optional as many extensions do not need to be configured previously to usage.

4.2.1 Proposed Implementation

By definition, the ITLingo Studio browser extension needs a Devtools page that allows the manipulation of a JSON file, which will contain all of the keywords and their corresponding X-Path counterparts. Devtools is a built in functionality provided by modern web browsers that allows for code debugging, reviewing, visualization of what is being loaded to the client browser, inspecting network requisitions, and others.

After that, a content script would need to be implemented to manage the click events on the tested web page, which will generate said X-Path mappings and callback the DevTools page with the result. The last item is implementing a background script that allows the communication between the DevTools UI Element and the injected content scripts. The resulting workflow can be described as in 4.3.

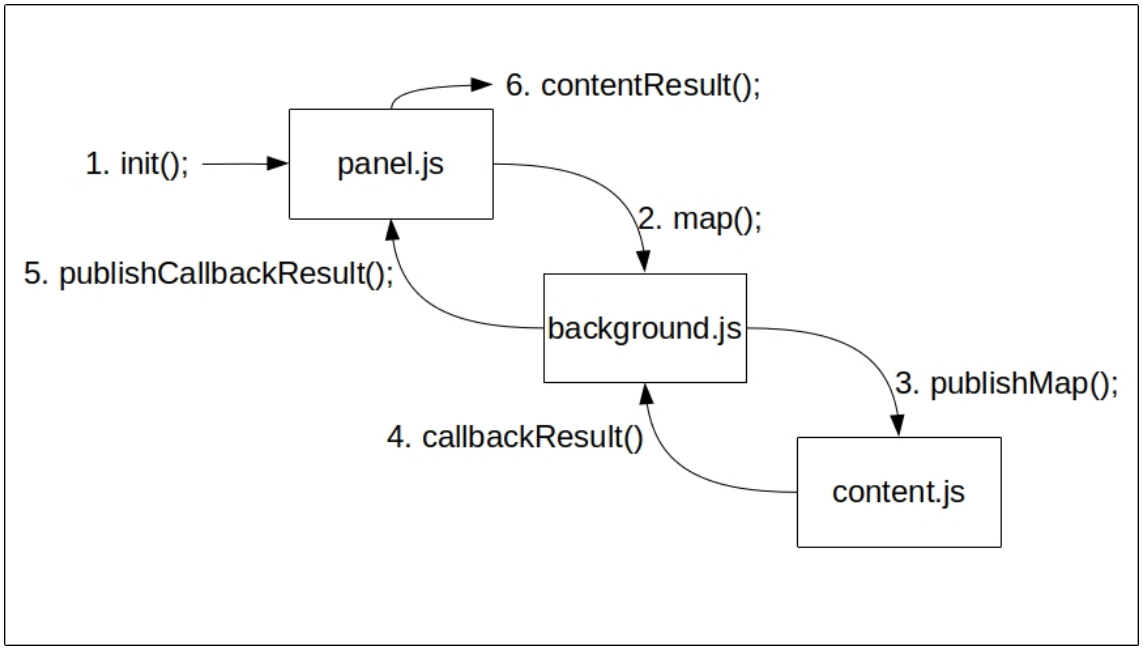

Figure 4.3: Workflow design to the chrome extension.

The process used to achieve keyword/component mapping starts by initializing every aspect of the UI Element, which is hosted in the browser's DevTools, which in turn allows for the correct mapping of UI elements to their counterparts in the Robot Framework script.

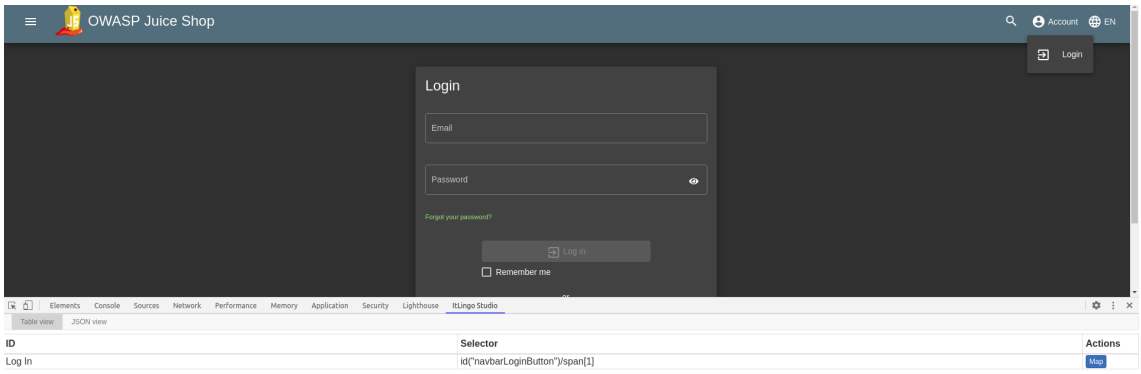

Figure 4.4: ITLingo Chrome extension mapping page action.

The UI Elements screen hosted in the DevTools can read an input JSON such as 4.7 and transform the items into a table that can trigger mapping calls to the background listener, which in turn relays the request to the currently active tab. This component's primary responsibilities are creating a connection to the background component called port and building all of the elements and events required to transform JSON files into tables correctly and vice-versa.

```
$(document).ready(function() {

    let _data = null;


    let tableLi = $('.table-view-li');
    let jsonLi = $('.JSON-view-li');
    let tableButton = $('.table-view-btn');
    let jsonButton = $('.JSON-view-btn');
    let tableView = $('#tableView');
    let jsonView = $('#jsonView');
    let jsonInput = $('.json-input');
    let tableTbodyLastChild = $('#selector-table > tbody:last-child');

    let mapBtn = '<td><button type="button" id="##id##" class="btn btn-primary btn-xs">Map</button></td>';
    let port = chrome.extension.connect({
        name: "PANEL_MAP"
    });

    //...
```

Figure 4.5: Initialization of the DevTools UI Element.

Port API elements that enable the message exchange between components and more than one port with different responsibilities can be created during a chrome extension. Ports are the main form of communication between extension components and can receive IDs, which is a useful way of controlling what connection is being created to the background script. In the ITLingo Studio browser extension, there was no need to make port name validations as only one port was needed

to manage the program's requests and responses.

```
chrome.extension.onConnect.addListener(
    listener: function (port : MediaQueryListEvent ) {
        port.onMessage.addListener( listener: function (message : MediaQueryListEvent ) {
            chrome.tabs.query( permissionDesc: {currentWindow: true, active: true}, function (tabs) {
                chrome.tabs.sendMessage(tabs[0].id, {action: "map", message: message});
            });
        });
        chrome.extension.onMessage.addListener( listener: function (message : MediaQueryListEvent , sender) {
            chrome.tabs.query( permissionDesc: {currentWindow: true, active: true}, function (tabs) {
                port.postMessage(message);
            })
        });
    });
```

Figure 4.6: The required code for the 'background.js' file basic functionality.

As seen in 4.6, the first listener that is added is responsible for receiving connection requests from UI Elements. Once a request is made, the port created in 4.5 is provided, and with this reference, the background script proceeds to create two listeners. The first listener handles all mapping events and messages and then forwards them to the content component attached in the active tab. This first step receives a message that contains an ID that represents the keyword that is being mapped. The second listener handles the response that is published by the content component in the active tab. It also receives a message containing the keyword ID that was previously passed and contains the mapping done in the form of an X-Path.

```
{
    "selectors": [
        {
            "id": "Log In",
            "selector": "id(\"mat-expansion-panel-header-0\")/span[1]/mat-panel-title[1]"
        }
    ]
}
```

Figure 4.7: JSON resulting from the manual mapping.

The content component is the last script that is executed by the extension. As mentioned above, the browser is injected onto the loading page and can run scripts based on the active tab. Every new tab injects a new instance of such a script.

Once a mapping event is published by the DevTools UI Element and relayed through by the Background Component, the Content component subscribes an event to the root of the page that handles the click of every element in the HTML. Once an element is clicked, a function can extract X-Path from the click event's target element and return it to the primary caller. Once this step is concluded, the Content component unsubscribes the click event from the page and returns the message with the ID and the X-Path to the Background component, which will relay it back to the DevTools UI Element component.

```
function loadData() {

    if(_data != null) {
        let value = JSON.stringify(_data, replacer: undefined, space: "\t");
        window.localStorage.setItem('json-value', value);
        jsonInput.html(value);
    }
    else {
        let value = window.localStorage.getItem( key: 'json-value');
        jsonInput.html(value);
        _data = validateJSON(value)
    }
}

function reloadTable() {
    loadData();
    tableTbodyLastChild.empty();

    if(_data == null)
        return;

    let selectors = _data.selectors;


    for(let i = 0; i < selectors.length; i++) {

        let btnId = 'btn-map-' + i.toString();

        let btn = mapBtn.replace( searchValue: '##id##', btnId);

        tableTbodyLastChild.append(
            '<tr>' +
                `<td>${selectors[i].id}</td>` +
                `<td>${selectors[i].selector}</td>` +
                `${btn}` +
            '</tr>');

        $('#' + btnId).click(map);
    }
}
```

Figure 4.8: DevTools UI Element functions that are responsible for updating the view.

Once the DevTools UI Element receives the response message, a new function is executed which updates the JSON and refreshes the table that fired the mapping event resulting in 4.4

4.3 Web Based ITLingo Studio

This implementation's original idea was to create a web application that implemented the code editing through three proposed web editors (Orion, Ace or CodeMirror) [14, 12] by taking advantage of the pre-existing capabilities of the Xtext Framework to generate web compatible DSL projects. Although this approach has its advantages, it would require writing an entire IDE from scratch, so it was discarded in favor of the recently released and multi-functional IDE called Eclipse Theia.

4.3.1 Introduction to Eclipse Theia IDE

Eclipse Theia is a browser-based IDE platform that supports all of the latest programming languages, and its extension capacities provide the capability of community created extensions to allow support to many other languages and technologies [42, 11].

By leveraging cloud and container technologies, it can allow for the project to be scaled beyond physical machines, simplify sharing and collaboration in a project, centralize control and confidentiality of the developed source code, and more [42, 11]. It still faces many challenges that need to be addressed to grow, such as bad network connection, integration with local operating systems, and different tools to perform many tasks. However, it has already reached a mature stage in its development cycle for the time being used in many day-to-day project environments [42, 11].

4.3.2 The ITLingo Studio Theia extension

In [42, 43] it is implemented a solution that allows using the improvements implemented in Section 4.1 it is possible to integrate the Xtext Language Server as a JAR to an Eclipse Theia extension that allows writing, validating, and highlighting written specifications and after it is done the language server is capable of executing all code generation functions normally.

This process is that the Language Server is executed in the background as a java application. Then the Theia connects to this background process through a web-socket that communicates the written specification in a two-way channel to correctly execute all of the required functions for the language to be correctly written and validated [42, 43].

Chapter 5

Validation

The objective of this research attempted to establish two main points. If the ITLingo Studio tool can execute complex automated test cases based on keyword mapping, which was not very explored until now, and if the improvements previously implemented have a performance gain in general in the tool.

To confirm that the improvements designed in the last phase are an actual improvement to the previous workflow [24] and are viable, an experiment that can leverage the new improvements was designed. The chosen application was OWASP - Juice Shop (OJS). OJS is a mock e-commerce application developed by OWASP. It features a full workflow of regular e-commerce sites, but the main idea behind this application has multiple security flaws that software security professionals and researchers can exploit.

OJS presents itself as an excellent test-bed for automation and security testing and helps uncover improvement points on the ITLingo Studio. Some improvement points are already implemented during this research, but others were not urgent and are not considered.

As the main idea behind the OJS application is security testing, the usage of its capabilities also raised a new line of questioning in this research: **Is it possible to perform security testing using the ITLingo Studio [27]?** This line of thought, although limited to input level security testing was promising and will be further discussed in **section 5.2**.

5.1 Functional Aspects

As demonstrated in **Chapter 4**, the improvements that implemented to the ITLingo Studio platform and software were not related to the requirements specification and engineering phase of the process, so the step of writing the entities and requirements through RSL were the same and for the remains of this research, the time to generate specs of a determined application remain unaltered.

Six entities have been mapped, and as a result of this process, three use cases were devised that can cover most of the platform's functionality and provides a complex set of interactions that are going to be tested against the ITLingo Studio testing capabilities. The use cases were **Sign-up**, **Sign-in**, and **Purchase**. The first two use cases are common in most of the present web

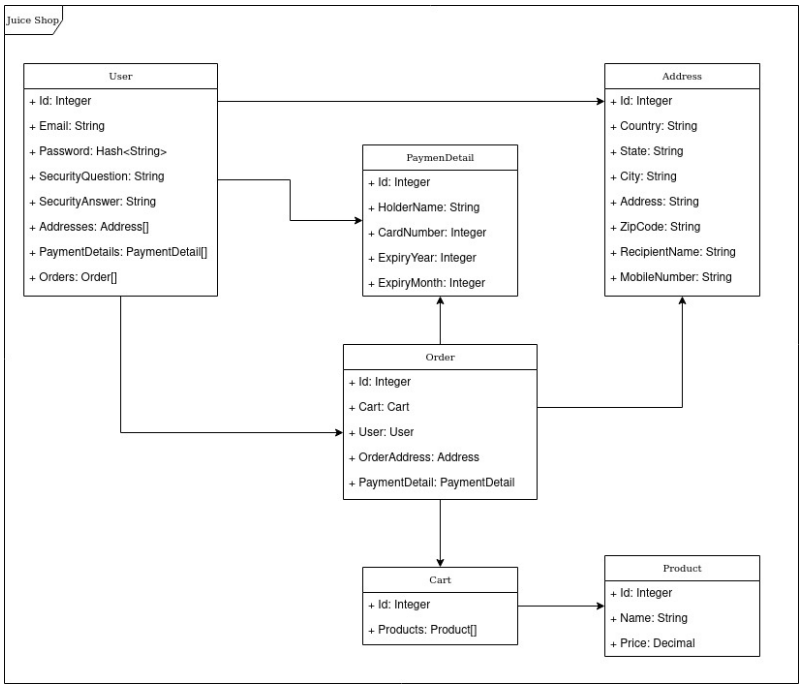

Figure 5.1: Class diagram depicting the entities of the OJS system.

applications, and particularly for the e-commerce platform, performing a purchase is one of the main and most actions a regular user can perform in such systems.

5.1.1 Sign-up

The '*Sign-up*' use case serves as a starting point to the testing process. This use case is responsible for the actions of a visitor who intends to create an account in the fictitious e-commerce.

```
1  DataEntity e_User "User" : Master [
2    attribute ID "ID" : Integer [isNotNull isUnique]
3    attribute email "Email" : Regex [isNotNull isUnique]
4    attribute password "Password" : Regex [isNotNull isEncrypted]
5    attribute securityQuestion "SecurityQuestion" : Text [isNotNull]
6    attribute securityAnswer "SecurityAnswer" : Text [isNotNull]
7    primaryKey (ID)]
8
9  Actor aU_User "User" : User [description "User uses the system"]
10
11 UseCase uc_1_SignUp "SignUp" : EntityCreate [
12   actorInitiates aU_Visitor
13     dataEntity e_User
14     description "Visitor creates an user account."]
```

Listing 5.1: RSL specification of data entity User and Sign-up use case

As mentioned above, the Sign-up use case is the entry point in this platform as no other actions can be performed before a visitor is adequately registered and logged in. This is a small use case and serves as the first interaction to the OJS.

The derived test script then requires that nine keywords are mapped in the web page. Of those nine keywords, eight can be considered directly related to the web page, and one is a general data privacy disclaimer and is accounted in all of the tests.

```
UseCaseTest t_ec_1_SignUp "SignUp" : Valid [
  useCase uc_1_SignUp actorInitiates aU_Visitor
  description "As a Visitor I want to sign up"
  variable user [
    attribute t_uc_1_email: String
    attribute t_uc_1_password: String
    attribute password_confirmation: String
    attribute t_uc_1_answer: String
    attribute success_message: String
  ]
  testScenario SignUp :Main [
    isConcrete
    variable user withValues (
        | user.t_uc_1_email | user.t_uc_1_password
        | user.password_confirmation | user.t_uc_1_answer
        | user.success_message +|
        | "fff@fff.com" | "12345"
        | "12345" | "abc123"
        | "Registration completed successfully. You can now log in." +|
    )
    step s1:System_Execute:OpenBrowser
            ["The system opens the browser in https://juice-shop.herokuapp.com/#/register"]
    step s2:Actor_CallSystem:Click element('t_uc_1_Dismiss')
            ["The Visitor clicks the 'Dismiss' button"]
    step s3:Actor_PrepareData:PostData readFrom user.t_uc_1_email
            ["The Visitor writes the email"]
    step s4:Actor_PrepareData:PostData readFrom user.t_uc_1_password
            ["The Visitor writes the password"]
    step s5:Actor_PrepareData:PostData readFrom user.password_confirmation
            ["The Visitor writes the password confirmation"]
    step s6:Actor_CallSystem:Click element('SecurityQuestion')
            ["The Visitor focuses the security question dropwdown"]
    step s7:Actor_CallSystem:Click element('SecurityAnswer')
            ["The Visitor selects the desired security answer"]
    step s8:Actor_PrepareData:PostData readFrom user.t_uc_1_answer
            ["The Visitor writes the security answer"]
    step s9:Actor_CallSystem:Click element('t_uc_1_Confirm')
            ["Submit the data"]
    step s10:System_Execute:Check elementOnScreen(text user.success_message)
            ["'Success toast message' appears"]
  ]
]
```

Listing 5.2: Test specification of Sign-up use case

After the Use case and test are correctly specified, new files are generated which contains the JSON that is used to write the keyword mapping to XPath and later interpreted into a Python script which contains the variables. The last file to be generated is the Robot Framework file itself, which, as mentioned in chapter 4, has undergone some modifications to improve the usability of the tests in general. Mainly these changes involve the implementation of sleep timers between every action in the script, which can be useful in scenarios where the tester is not running the application locally. And the separation of the variables to the python file which helps the robot to suffer fewer modifications during the phase of development of requirements.

```
SignUp-Test_1
  [Documentation]     As a Visitor I want to sign up
  open browser  https://juice-shop.herokuapp.com/#/register  ${Browser}
  Sleep  2s
  Click element  ${t_uc_1_Dismiss}
  Sleep  2s
  input text   ${t_uc_1_email}   ${t_uc_1_email1}
  Sleep  2s
  input text   ${t_uc_1_password}   ${t_uc_1_password1}
  Sleep  2s
  input text   ${password_confirmation}   ${password_confirmation1}
  Sleep  2s
  Click element  ${SecurityQuestion}
  Sleep  2s
  Click element  ${SecurityAnswer}
  Sleep  2s
  input text   ${t_uc_1_answer}   ${t_uc_1_answer1}
  Sleep  2s
  Click element  ${t_uc_1_Confirm}
  Sleep  2s
  Page Should Contain Element ${SignUp_PointAndClick}  ${success_message1}
```

Listing 5.3: Robot script derived from the test specification in the RSL file

After the generation of the files mentioned above, the testes should head to the browser where the keyword mapping step is executed. As mentioned above, the sign-up script requires the mapping of 9 keywords into UI elements, and the main way of doing so is using the ITLingo Studio chrome extension, that is presented in 4.

Once the keywords are correctly mapped inside the JSON file, the python script is automatically updated with the correct mapping which allows the test to be executed and consequently validated. If for some reason a mapping is not correct, the tester is able to manually fix it and re execute the test.

5.1.2 Sign-in

The idea of the “Sign-In“ test is to check if a previously registered user can sign in according to the documentation by providing the e-mail and their respective password. These test results are a success even though it presented the same issues as the previous test. A successful login attempt is made, and the user is correctly authenticated.

This use case is crucial as it has to be reused in the subsequent use cases and actions as every test executed by ITLingo Studios is run in a clean environment.

5.1.3 Purchase

The ’*Purchase*’ use case examines the ITLingo Studio capabilities and new functionalities in a more complex scenario, such as the entire process required to perform the purchase of an item sold in the fictitious e-commerce platform.

This test is the most extended and most complex in the suite and requires interaction with multiple elements, such as input, select, radio button, button, and other HTML elements.

Listing 5.4 presents a snippet of the test script that in natural language translates to:

A user interested in buying juice enters the Juice Shop website and logs in with their email and password. After performing this action, the user is redirected to the main screen of the application, selecting which products they want to buy. Once satisfied with the selection, the user goes to the basket where he can checkout. Once in the checkout process, the user must register a new address or select an existing one from the list, select the shipping method and, finally, register or select a payment method from the list. After completing these steps, the user finalizes his transaction and waits for the delivery.

```
step s11:Actor_CallSystem:Click element('t_uc_3_NewAddress')
    ["The User Adds a new Address so that his products can be delivered"]
step s12:Actor_PrepareData:PostData readFrom order.t_uc_3_country
    ["The User writes the country"]
step s13:Actor_PrepareData:PostData readFrom order.t_uc_3_addressName
    ["The User writes an alias to this address"]
step s14:Actor_PrepareData:PostData readFrom order.t_uc_3_mobileNumber
    ["The User writes his phone number"]
step s15:Actor_PrepareData:PostData readFrom order.t_uc_3_zipCode
    ["The User writes the ZIP code"]
step s16:Actor_PrepareData:PostData readFrom order.t_uc_3_address
    ["The User writes the address"]
step s17:Actor_PrepareData:PostData readFrom order.t_uc_3_city
    ["The User writes the city"]
step s18:Actor_PrepareData:PostData readFrom order.t_uc_3_state
    ["The User writes the state"]
step s19:Actor_CallSystem:Click element('t_uc_3_SubmitAddress')
    ["The User submits the input address"]
step s20:Actor_CallSystem:Click element('t_uc_3_ChooseAddress')
    ["The User selects the previously added address"]
step s21:Actor_CallSystem:Click element('t_uc_3_Continue1')
    ["The User continues the transaction"]
```

Listing 5.4: Snippet of the test specification related to the creation of a new address for the Purchase use case

The test script required to achieve this test requires 23 keywords that need to be addressed (Dismiss discarded) before running the generated Robot Framework script as depicted on table 5.1. The steps require the interaction with many different components of the application. As mentioned, Juice Shop is written in Angular, and one of the main concepts of such a framework is the componentization of its views. By testing the transitions between those views, new elements are going to be plotted and added through custom HTML attributes, thus making an excellent subject for the ITLingo Studio in many levels, such as the web scrapping extension and the mapping itself, which later will be handled by Robot Framework and Selenium Web Driver.

The snippet presented in listing 5.4 presents the test steps required to create a new address and select it. In this context, the script must create a new valid e-mail address containing text inputs, number inputs, a text area, and in the and one or more radio buttons. These elements need to be correctly filled to the form to be valid and the test to proceed.

The test resulted in a successful purchase attempt, which suggests that ITLingo Studio can handle not only big test scenarios but is also capable of testing modern and complex web applications like OJS, that uses Googles Angular Framework.

Table 5.1: Test details by view

View	# Test Steps
Home page	4
Sign-in	3
Basket	1
Checkout (address)	11
Checkout (shipping)	2
Checkout (payment)	2

5.2 Security Aspects

The documentation of OWASP Juice Shop [22] to identify and understand the involved exploits. Most of its exploits referred to the following vulnerabilities: SQL Injection, XSS, Vulnerable Components, Sensitive Data Exposure, Broken Authentication, Broken Access Control.

Many of these exploits require tools that have to be combined with ITLingo Studio, including features such as browser developer mode, use external programs to inspect requests and penetration testing tools. This study explores the following vulnerabilities: SQL injection and XSS based on UI inputs.

5.2.1 Multiple Invalid Login Attempts Test Case

This test aims to check, by repetition, any protection against multiple incorrect login attempts such as IP blocking, account locking exists. This attempt tries to emulate the incorrect submission of a username/password combination before attempting to log in with the correct password to check if the user/account was blocked. The repeated submission of requests to a determined application could also be identified as a potential Denial of Service attack. An application that has protection against such attacks could also block a malicious user after several login attempts.

This test's main objective is to check if, after 3 to 5 attempts of requests with invalid passwords, the system will present any lock or captcha to block malicious attempts to sign-in. This test results showed that the web application has no countermeasures against brute force attacks as after five attempts, it was possible to keep retrying until the password was correct.

```
1  UseCaseTest st_uc_1_BruteForce "SignInBruteForce": Valid [
2    useCase suc_1_BruteForce_SignIn actorInitiates aU_Cracker
3    description "As a Cracker I want to brute force sign in"
4      variable user [
5      attribute email: String
6      attribute password: String
7      attribute wrong_password: String
8    ]
9    testScenario BruteForceSignIn : Main [
10     isConcrete
11     variable user  withValues (
12     | user.email                    | user.wrong_password | user.password +|
13     | "wypizsbdbmiazqybiz@awdrt.net" | "q1w2e3r4t5"        | "q1w2e3r4"    +|
```

```
14    )
15      step s1:System_Execute:OpenBrowser
16                ["The system opens the browser in https:juice-shop.herokuapp.com/#/"]
17    step s2:Actor_CallSystem:Click element('Dismiss')
18                ["The Visitor clicks the 'Dismiss' button"]
19    step s3:Actor_CallSystem:Click element('Account')
20      step s4:Actor_CallSystem:Click element('SignIn')
21            ["The Visitor clicks on the 'SignIn' button"]
22    step s5:Actor_PrepareData:PostData readFrom user.email
23            ["The Visitor writes the email"]
24    step s6:Actor_PrepareData:PostData readFrom user.wrong_password
25            ["The Visitor writes the wrong password"]
26    step s7:Actor_CallSystem:Click element('Confirm_SignIn')
27            ["The Visitor clicks on the 'Sign In' button"]
28    step s8:Actor_CallSystem:Click element('Confirm_SignIn')
29            ["The Visitor clicks on the 'Sign In' button"]
30    step s9:Actor_CallSystem:Click element('Confirm_SignIn')
31            ["The Visitor clicks on the 'Sign In' button"]
32    step s10:Actor_CallSystem:Click element('Confirm_SignIn')
33            ["The Visitor clicks on the 'Sign In' button"]
34    step s11:Actor_CallSystem:Click element('Confirm_SignIn')
35            ["The Visitor clicks on the 'Sign In' button"]
36    step s12:Actor_PrepareData:PostData readFrom user.password
37            ["The Visitor writes the correct password"]
38    step s13:Actor_CallSystem:Click element('Confirm_SignIn')
39            ["The Visitor clicks on the 'Sign In' button"]
40    step s14:Actor_CallSystem:Click element('My_Account')
41            ["The Visitor clicks on the 'My Account' button"]
42    step s15:System_Execute:Check elementOnScreen(text user.email)
43            ["'MyAccount' appears with the user name"]
44  ]
45 ]
```

Listing 5.5: Multiple invalid login attempts test case (RSL spec).

The result of this first set of tests showed that, although it is possible to perform such validations using RSL and ITLingo Studio, this may be a very repetitive and time-consuming task, as the language currently does not support iteration operators, thus requiring the manual implementation of each invalid sign-in attempt. In the Juice Shop application test case, after five attempts with wrong passwords, the test login with a correct password and was successfully authenticated, and thus, this shows that the Juice Shop application has no protection against brute force sign-in attacks or DDOS attacks.

5.2.2 SQL Injection attack - Abuse Login as Administrator

This test aims to check if it is possible to execute an SQL injection attack on the system. It requires some previous knowledge of the system architecture, but in general, this knowledge is available.

Spec. 5.6 shows an RSL specification of the “Abuse Login as Administrator" test case. This test is a simple example of an SQL injection attack. The goal of this test is to break the login query by sending a particular sequence of characters (i.e., the string "’ OR 1=1–") that returns true, and that allows to login with the first user of the application’s database: that happens to be the “administrator" user! With this test, it was possible to detect a vital system vulnerability, since it is possible to incorrectly login as an administrator and, thus, log in with higher privileges accessing information that should not be supposed to.

This test resulted in a successful login as an administrator, in a real scenario that would allow an attacker to execute actions that should not be available to regular users.

```
UseCaseTest st_uc_1_SQLInject "" : Valid [
  useCase suc_2_SQLi_Admin_User actorInitiates aU_Cracker
  description "As a Cracker I want to login as Administrator using SQLi"
  variable query [
    attribute login_query: String
    attribute password_query: String
    attribute admin_email: String
        attribute blabla: String
  ]
  testScenario SQLinjectionAdministratorSignIn : Main [
    isConcrete
    variable query withValues (
      | query.login_query | query.password_query | query.admin_email +|
      | "' OR 1=1--" | "anything" | "admin@juice-sh.op" +|
    )

      step s1:System_Execute:OpenBrowser ["The system opens the browser in https:juice-shop.herokuapp.com/#/"]
    step s2:Actor_CallSystem:Click element('Dismiss') ["The Visitor clicks the 'Dismiss' button"]
    step s3:Actor_CallSystem:Click element('Account')[""]
      step s4:Actor_CallSystem:Click element('SignIn') ["The Visitor clicks on the 'SignIn' button"]
    step s5:Actor_PrepareData:PostData readFrom query.login_query ["The Cracker writes the query"]
    step s6:Actor_PrepareData:PostData readFrom query.password_query ["The Cracker writes random data"]
    step s7:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the 'Sign In' button"]
    step s8:Actor_CallSystem:Click element('My_Account') ["The Visitor clicks on the 'My Account' button"]
    step s9:System_Execute:Check elementOnScreen(text query.admin_email) ["'MyAccount' appears with the user name"]
  ]
]
```

Listing 5.6: Multiple invalid login attempts test case (RSL spec).

5.2.3 Embed IFrame which calls Another Website on Screen

This test consists in creating an address that would return a specific set of characters, then by inputting an IFrame that references that address and submit the search query to test if the system renders the input data as HTML.

This test aims to check if it is possible to fill in an input field with an IFrame redirecting to a new location that does not belong to the application under test. Indeed, it was possible to confirm this vulnerability since after the search input form, the system redirected to the new location. However, this test required the local execution of the OWASP Juice Shop as it is needed to access a local API.

5.3 Discussion

This section goal is to discuss the results presented in the examples presented above, which are separated in two main major areas, requirements engineering, and security testing, and answer the questions raised at the beginning of this chapter.

Is ITLingo Studio capable of handling complex scenarios of modern web applications? Recently, Maciel et al. presented a concept that uses the ITLingo Studio to test a website similar to OJS called Automation Practice. Much like OJS, this website is a mock e-commerce application, with the main difference being that it is not coded with security flaws in mind. At a technical level, Automation Practice is developed using PHP and necessary Javascript, which characterizes an older web application at its core.

The first trial runs executed on the OJS application did not consider the improvements that would later integrate into the ITLingo Studio. Although it could run complex scenarios in the Automation Practice web page, the mappings' quality decreases drastically when the same tests run on OJS. This situation occurred because, in Angular applications, IDs and other types of attributes useful on CSS mapping may be dynamic.

This issue required a new line of approach where XPath mappings were to be used instead of CSS mappings, and with minor tweaks directly on the JSON mapping file, the ITLingo Studio was then capable of executing all of the complex test scenarios presented in OJS. In this scenario, the answer to the question raised in this section's title is **yes, but with limitations** as a new manual mapping technique that fills the gap between legacy and modern code.

Did the proposed improvements bring productivity gains? In a quick recap of the changes introduced to the ITLingo Studio and presented in 4, the code generator is now able to handle all of the processes of creating JSON files, Robot Framework files, and the brand new Python variables script automatically. Also, the generated source code will be placed inside the folder where the original RSL is. These modifications allow for a better context separation as a requirements engineer may be working on more than one RSL file in different contexts. The other improvements include a new web scraping browser extension and the integration to the Web IDE Eclipse Theia, which allows faster and seamless integration of the DSL to the IDE.

In order to assert what was the performance gain between the older version and the more recent version of the ITLingo Studio the three use cases defined above were reused to generate the files that need to be mapped and executed. This process generated around forty variables that need mapping.

The original workflow as presented in [24] presents itself in six steps as depicted below:

1. Write RSL specification;
2. Curate variables in the automatically generated Robot Framework file;
3. Map the variables using the Web Scraper extension [44];
4. Export the mapped variables into a JSON file;
5. Execute the *mapping.java* file which parses the JSON file into a XML;
6. Execute the *replace.java* file which replaces the keyword mapping present in the XML into the Robot Framework file.

These steps are handled in a different program such as the Eclipse plugin for the ITLingo Studio, The Web Scraper browser extension [44], a Java IDE of the requirements engineer‘s choice, and finally, a terminal emulator that allows the user to run the script.

Some of the issues around this approach are that normally Requirements Engineers don’t have available tools such as programming IDEs required to run a few steps of the process. The improvements described in **Chapter 4** are designed specifically to fix these issues and improve the usability and performance of the ITlingo Studio. That being said, the process was reduced to a total of two steps:

1. Write RSL specification;
2. Map the variables to their counterpats in the Python script using the new ITlingo Studio Web Scraper extension.

This reduction of steps was possible through the automation of a significant part of the steps that are executed semi-automatically, and as such, result in a reduction in the time needed to carry out the entire mapping process necessary to reach the point where it is possible to perform automated tests of approximately four minutes and twenty-three seconds (around 43% of overall improvement) as illustrated in the **Chart 5.1**.

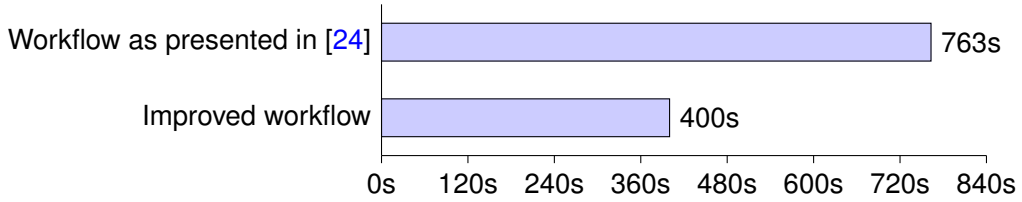

Chart 5.1: Total time needed to get a Robot Script ready to execute in seconds.

In both workflows, the most time-consuming task is the step of mapping keywords to screen elements, as they cannot be correctly automated and require simultaneous user interaction with the application under testing. The answer to the question raised in this section can be made in two parts:

Map keywords to elements - In the original experiment it was required around eight minutes and thirty seconds to map all of the forty curated keywords, while in the improved version of ITLingo Studio it took around five minutes and forty five seconds. Although it is a good improvement (around 28% less time required to end the task), it is not too big, as this is the most important and error prone step of the process.

Other tasks - These are the tasks that are now almost entirely automated in the new version of ITLingo Studio, so their execution time has dramatically reduced. Thus, while in the previous version, it takes four minutes and forty-two seconds to perform these other tasks, in the new version, only fifty-five seconds are needed to perform them (around 80% less time required to perform these steps), and these are only limited to the transfer of information between ITLingo Studio IDE and its web extension.

Thus, it is possible to conclude that the productivity gains are somewhat considerable and take a great deal of work from the requirements Engineers, thus allowing them to focus on what is essential, which is the correct keyword mapping to their counterparts in the application UI.

What are the security testing capabilities of the ITLingo Studio as it is now? Based on the results achieved in **Section 5.2** is possible to conclude that the proposed approach (with tools like ITLingo Studio and Robot Framework) can be adopted to test security vulnerabilities. ITLingo Studio is only capable of performing interactions and inputs directly to an application's UI, thus limiting the range of security tests that are possible. With the implementation of new features into the RSL language, new types of tests could be allowed, but for now, only security tests related to inputs performed directly in the application are possible.

So, the answer to the question raised by this section is yes, but at the moment, it needs to be extended and combined with additional functionalities and other cyber-security specific tools as the current testing possibilities are minimal.

Chapter 6

Conclusion

This research provides a new form of interacting with the ITLingo Studio and its RSL language, which was not possible before. Some of the improvements may be transparent to the end-user in a day-to-day base, but the web IDE utilization raises the tool to a new level of maturity, as it is more easily extensible than the original Eclipse plugin.

6.1 Final Discussion

Maciel et. al. demonstrated in [24] that the improvements developed are focused on the generation of test cases based on the specification model written by a Requirements Engineer. Although test case generation and automation is accomplished, it has not been tested in scenarios that can be considered more progressive, such as using the tool to test Angular applications, which raises the overall complexity by a lot and achieve the goal of testing web applications. In a more broad sense, it was required to shift the implementation from CSS mapping to XPath mapping, which helped to reduce the necessary effort, except in some moments, was that the requirements engineer needed to correct some mapping that needed a greater specificity. In a test with approximately thirty-six keywords, two interventions are needed.

The need to create XPath mappings leads to implementing a new chrome extension that focuses only on mapping elements to their related XPath. Furthermore, by taking leverage the code generator embedded to the ITLingo Studio Xtext project, it is possible to fully automate the keywords mapping and generation into a JSON file, which then is imported into the new browser extension, thus almost eliminating the need to perform tasks that do not need human interaction.

With this improvements, the Requirements Engineer can focus on writing the requirements, the tests, and after those tests are ready, importing the JSON that is generated through the interpretation of such test scripts by the ITLingo Studio into the Web Browser and then performing the keyword mapping.

It is also possible to conclude that although with some limitations, the ITLingo Studio has the capabilities to execute different types of testing, such as Security testing, that even though has

limitations to it, some could be overcome through the implementation of new features that are created with this kind of testing in mind.

6.2 Future Work

This dissertation opens new forms of improving ITLingo Studio by providing new tools such as the Web IDE and the Web Browser Extension. By leveraging both applications' capabilities, some of the new features that could be accomplished and would benefit the end user are:

- Evaluate the possibility of using websocket technologies to automatically communicate the file structure from the ITLingo Web IDE to the Chrome extension, this way the Requirements Engineer would not be required to manually copy data from one side to the other.
- XPath mapping can be achieved in many different ways, and this research proposed an approach based on finding elements through the root path and then counting them. Other algorithm options could benefit the application in a manner of having different options to achieve element mapping and this way help eliminate the need for human intervention on point-and-click UI mapping. Selenium IDE, for example, besides XPath mapping offers also CSS mappings, and each of these with more then one option to be used [16].

In the RSL Language, some new features could help security testing, such as iterative elements in the test scripts would be benefit, for example, test cases that need to make repetitive tasks, such as tests focused on brute forcing an application or tests simulating DDOS.

Finally, some of these ideas, such as automated security testing, are explored to some extent and prove to be very relevant to the ITLingo Studio tool to improve development applications' quality. Besides the possibility of being further explored, other types of tests could also be evaluated, such as performance and robustness testing.

Appendix A

Full Specification of the Juice Shop

```
Package WebApp


/*******************************************************
   System definition
*******************************************************/

System MyStoreWebApp "Juice Shop (Application Level)" : Application [ isFinal
  description "Juice Shop is a fictitious web store for penetration testing
          testing purposes that can be used to automation testing approaches too"]


/*******************************************************
   DataEntities view
*******************************************************/

DataEntity e_User "User" : Master [
  attribute ID "ID" : Integer [isNotNull isUnique]
  attribute email "Email" : Regex [isNotNull isUnique]
  attribute password "Password" : Regex [isNotNull isEncrypted]
  attribute securityQuestion "SecurityQuestion" : Text [isNotNull]
  attribute securityAnswer "SecurityAnswer" : Text [isNotNull]
  primaryKey (ID)
  description "Users"
]

DataEntity e_Address "Address" : Master [
    attribute ID "ID": Integer [isNotNull isUnique]
    attribute country "Country": String [isNotNull]
    attribute recipientName "RecipientName": String [isNotNull]
    attribute mobileNumber "MobileNumber": String [isNotNull]
    attribute zipCode "ZipCode": String [isNotNull]
    attribute address "Address": String [isNotNull]
    attribute city "City": String [isNotNull]
    attribute recipientState "RecipientState": String [isNotNull]
  primaryKey (ID)
    description "Addresses"
```

```
38 ]
39
40 DataEntity e_PaymentDetail "PaymentDetail": Master [
41     attribute ID "ID": Integer [isNotNull isUnique]
42     attribute holderName "HolderName": String [isNotNull]
43     attribute cardNumber "CardNumber": String [isNotNull]
44     attribute expiryMonth "ExpiryMonth": Integer [isNotNull]
45     attribute expiryYear "ExpiryYear": Integer [isNotNull]
46   primaryKey (ID)
47     description "Payment Details"
48 ]
49
50 DataEntity e_Order "Order" : Master [
51     attribute ID "ID": Integer [isNotNull isUnique]
52     attribute cart "Cart": Integer [isNotNull isUnique]
53     attribute paymentDetail "PaymentDetail": Integer [isNotNull isUnique]
54     attribute user "User": Integer [isNotNull]
55   primaryKey (ID)
56     foreignKey e_Cart(cart)
57     foreignKey e_PaymentDetail(paymentDetail)
58     foreignKey e_User(user)
59     description "Orders"
60 ]
61
62 DataEntity e_Cart "Cart" : Master [
63     attribute ID "ID" : Integer [isNotNull isUnique]
64     attribute product "Product" :  Integer [isNotNull isUnique]
65     primaryKey (ID)
66     foreignKey e_Product(product)
67     description "Carts"
68 ]
69
70 DataEntity e_Product "Product" : Master [
71     attribute ID "ID" : Integer [isNotNull isUnique]
72     attribute productName "ProductName" : String [isNotNull]
73     description "Products"
74 ]
75
76 Actor aU_User "User" : User [description "User uses the system"]
77 Actor aU_Visitor "Visitor" : User [description "Visitor who sees the system"]
78 Actor aU_Cracker "Cracker" : User [description "Cracker who wants to break into system"]
79
80 UseCase uc_1_SignUp "SignUp" : EntityCreate [
81   actorInitiates aU_Visitor
82     dataEntity e_User
83     description "Visitor creates an user account."
84 ]
85
86 UseCase uc_2_SignIn "SignIn" : EntityRead [
87   actorInitiates aU_Visitor
88   dataEntity e_User
89   description "Visitor log-in."
90 ]
91
```

```
92 UseCase uc_3_Purchase : EntityCreate [
93     actorInitiates aU_User
94     dataEntity e_Cart
95     description "User performs a purchase"
96 ]
97
98 UseCase suc_1_BruteForce_SignIn "BruteForceSignIn" : EntityRead [
99   actorInitiates aU_Cracker
100  dataEntity e_User
101    description "Cracker brute force log-in."
102 ]
103
104 UseCase suc_2_SQLi_Admin_User "SQLinjectionAdminSignIn" : EntityRead [
105  isNegative
106  actorInitiates aU_Cracker
107  dataEntity e_User
108    description "Cracker uses sql injection to log-in as administrator."
109 ]
110
111 /*******************************************************
112   Use Case Tests
113 *******************************************************/
114
115 UseCaseTest t_ec_1_SignUp "SignUp" : Valid [
116  useCase uc_1_SignUp actorInitiates aU_Visitor
117  description "As a Visitor I want to sign up"
118  variable user [
119    attribute t_uc_1_email: String
120    attribute t_uc_1_password: String
121    attribute password_confirmation: String
122    attribute t_uc_1_answer: String
123    attribute success_message: String
124  ]
125  testScenario SignUp :Main [
126    isConcrete
127    variable user withValues (
128      | user.t_uc_1_email    |  user.t_uc_1_password | user.password_confirmation | user.
           t_uc_1_answer | user.success_message                        +|
129      | "fff@fff.com"        |  "12345"            | "12345"            | "abc123"         |
           "Registration completed successfully. You can now log in." +|
130    )
131    step s1:System_Execute:OpenBrowser ["The system opens the browser in https://juice-shop.
         herokuapp.com/#/register"]
132    step s2:Actor_CallSystem:Click element('t_uc_1_Dismiss') ["The Visitor clicks the '
         Dismiss' button"]
133    step s3:Actor_PrepareData:PostData readFrom user.t_uc_1_email ["The Visitor writes the
         email"]
134    step s4:Actor_PrepareData:PostData readFrom user.t_uc_1_password ["The Visitor writes the
          password"]
135    step s5:Actor_PrepareData:PostData readFrom user.password_confirmation ["The Visitor
         writes the password confirmation"]
136    step s6:Actor_CallSystem:Click element('SecurityQuestion')["The Visitor focuses the
         security question dropwdown"]
```

```
137     step s7:Actor_CallSystem:Click element('SecurityAnswer')["The Visitor selects the desired
            security answer"]
138     step s8:Actor_PrepareData:PostData readFrom user.t_uc_1_answer ["The Visitor writes the
            security answer"]
139     step s9:Actor_CallSystem:Click element('t_uc_1_Confirm')["Submit the data"]
140     step s10:System_Execute:Check elementOnScreen(text user.success_message) ["'Success toast
            message' appears"]
141   ]
142 ]
143
144 UseCaseTest t_uc_2_SignIn "SignIn" : Valid [
145   useCase uc_2_SignIn actorInitiates aU_Visitor
146   description "As a Visitor I want to sign in"
147       variable user [
148     attribute t_uc_2_email: String
149     attribute t_uc_2_accountEmail: String
150     attribute t_uc_2_password: String
151     ]
152   testScenario SignIn :Main [
153     isConcrete
154     variable user  withValues (
155       | user.t_uc_2_email  | user.t_uc_2_accountEmail   | user.t_uc_2_password  +|
156       | "ddd@ddd.com"    | "ddd@ddd.com"              | "12345"               +|
157     )
158       step s1:System_Execute:OpenBrowser ["The system opens the browser in https://juice-shop
              .herokuapp.com/#/login"]
159       step s2:Actor_CallSystem:Click element('t_uc_2_Dismiss') ["The Visitor clicks the '
              Dismiss' button"]
160     step s3:Actor_PrepareData:PostData readFrom user.t_uc_2_email ["The Visitor writes the
            email"]
161     step s4:Actor_PrepareData:PostData readFrom user.t_uc_2_password ["The Visitor writes the
            password"]
162     step s5:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
163     step s6:Actor_CallSystem:Click element('My_Account') ["The Visitor clicks on the 'My
            Account' button"]
164     step s7:System_Execute:Check elementOnScreen(text user.t_uc_2_accountEmail) ["'MyAccount'
            appears with the user name"]
165     ]
166 ]
167
168 UseCaseTest t_uc_3_Purchase "Purchase" : Valid [
169     useCase uc_3_Purchase actorInitiates aU_User
170     description "As a User I want to buy a product"
171       variable order [
172     attribute t_uc_2_email: String
173     attribute t_uc_2_password: String
174         attribute t_uc_3_country: String
175         attribute t_uc_3_addressName: String
176         attribute t_uc_3_mobileNumber: String
177         attribute t_uc_3_zipCode: String
178         attribute t_uc_3_address: String
179         attribute t_uc_3_city: String
180         attribute t_uc_3_state: String
```

```
181 ]
182   testScenario Purchase : Main [
183       isConcrete
184   variable order withValues (
185     | order.t_uc_2_email   | order.t_uc_2_password | order.t_uc_3_country | order.
          t_uc_3_addressName |  order.t_uc_3_mobileNumber | order.t_uc_3_zipCode | order.
          t_uc_3_address   | order.t_uc_3_city | order.t_uc_3_state  +|
186         | "ddd@ddd.com"        | "12345"               | "Portugal"           | "FEUP"
                                  | "999 999 999"              | "123456"             | "R.
              Dr. Roberto Frias" | "Porto"           | "Porto"             +|
187   )
188     step s1:System_Execute:OpenBrowser ["The system opens the browser in https://juice-shop
          .herokuapp.com/#/"]
189   step s2:Actor_CallSystem:Click element('t_uc_3_Dismiss') ["The User clicks the 'Dismiss'
        button"]
190   step s3:Actor_CallSystem:Click element('Account')[""]
191     step s4:Actor_CallSystem:Click element('SignIn') ["The User clicks on the 'SignIn'
          button"]
192   step s5:Actor_PrepareData:PostData readFrom order.t_uc_2_email ["The User writes the
        email"]
193   step s6:Actor_PrepareData:PostData readFrom order.t_uc_2_password ["The User writes the
        password"]
194   step s7:Actor_CallSystem:Click element('Confirm_SignIn') ["The User clicks on the 'Sign
        In' button"]
195       step s8:Actor_CallSystem:Click element('t_uc_3_Product') ["The User adds a product to
            his basket"]
196       step s9:Actor_CallSystem:Click element('t_uc_3_Basket') ["The User goes to the basket
            "]
197       step s10:Actor_CallSystem:Click element('t_uc_3_Checkout') ["The User wants to
            checkout his basket"]
198       step s11:Actor_CallSystem:Click element('t_uc_3_NewAddress') ["The User Adds a new
            Address so that his products can be delivered"]
199       step s12:Actor_PrepareData:PostData readFrom order.t_uc_3_country ["The User writes
            the country"]
200       step s13:Actor_PrepareData:PostData readFrom order.t_uc_3_addressName ["The User
            writes an alias to this address"]
201       step s14:Actor_PrepareData:PostData readFrom order.t_uc_3_mobileNumber ["The User
            writes his phone number"]
202       step s15:Actor_PrepareData:PostData readFrom order.t_uc_3_zipCode ["The User writes
            the ZIP code"]
203       step s16:Actor_PrepareData:PostData readFrom order.t_uc_3_address ["The User writes
            the address"]
204       step s17:Actor_PrepareData:PostData readFrom order.t_uc_3_city ["The User writes the
            city"]
205       step s18:Actor_PrepareData:PostData readFrom order.t_uc_3_state ["The User writes the
            state"]
206       step s19:Actor_CallSystem:Click element('t_uc_3_SubmitAddress') ["The User submits
            the input address"]
207       step s20:Actor_CallSystem:Click element('t_uc_3_ChooseAddress') ["The User selects
            the previously added address"]
208       step s21:Actor_CallSystem:Click element('t_uc_3_Continue1') ["The User continues the
            transaction"]
209       step s22:Actor_CallSystem:Click element('t_uc_3_shippingMethod') ["The User selects
            the desired shipping method"]
```

```
210         step s23:Actor_CallSystem:Click element('t_uc_3_Continue2') ["The User continues the
                transaction"]
211         step s25:Actor_CallSystem:Click element('t_uc_3_ChooseCreditCard') ["The User selects
                 the previously added card"]
212         step s26:Actor_CallSystem:Click element('t_uc_3_Continue3') ["The User continues the
                transaction"]
213         step s27:Actor_CallSystem:Click element('t_uc_3_PlaceOrder') ["The User places the
                order"]
214     step s28:System_Execute:Check elementOnScreen(text "Thank you for your purchase!") ["The
            order has been placed appears"]
215     ]
216 ]
217
218  /*******************************************************
219    Security Tests
220  *******************************************************/
221
222 /*UseCaseTest st_uc_1_BruteForce "SignInBruteForce": Valid [
223   useCase suc_1_BruteForce_SignIn actorInitiates aU_Cracker
224   description "As a Cracker I want to brute force sign in"
225     variable user [
226     attribute email: String
227     attribute password: String
228     attribute wrong_password: String
229   ]
230   testScenario BruteForceSignIn : Main [
231     isConcrete
232     variable user  withValues (
233       | user.email            | user.wrong_password   | user.password   +|
234       | "wypizsbdbmiazqybiz@awdrt.net"  | "q1w2e3r4t5"      | "q1w2e3r4"      +|
235     )
236       step s1:System_Execute:OpenBrowser ["The system opens the browser in https:juice-shop.
              herokuapp.com/#/"]
237     step s2:Actor_CallSystem:Click element('Dismiss') ["The Visitor clicks the 'Dismiss'
            button"]
238     step s3:Actor_CallSystem:Click element('Account')[""]
239       step s4:Actor_CallSystem:Click element('SignIn') ["The Visitor clicks on the 'SignIn'
              button"]
240     step s5:Actor_PrepareData:PostData readFrom user.email ["The Visitor writes the email"]
241     step s6:Actor_PrepareData:PostData readFrom user.wrong_password ["The Visitor writes the
            wrong password"]
242     step s7:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
243     step s8:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
244     step s9:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
245     step s10:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
246     step s11:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
247     step s12:Actor_PrepareData:PostData readFrom user.password ["The Visitor writes the
            correct password"]
```

```
248     step s13:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
249     step s14:Actor_CallSystem:Click element('My_Account') ["The Visitor clicks on the 'My
            Account' button"]
250     step s15:System_Execute:Check elementOnScreen(text user.email) ["'MyAccount' appears with
             the user name"]
251   ]
252 ]
253
254 UseCaseTest st_uc_1_SQLInject "" : Valid [
255   useCase suc_2_SQLi_Admin_User actorInitiates aU_Cracker
256   description "As a Cracker I want to login as Administrator using SQLi"
257   variable query [
258     attribute login_query: String
259     attribute password_query: String
260     attribute admin_email: String
261         attribute blabla: String
262   ]
263   testScenario SQLinjectionAdministratorSignIn : Main [
264     isConcrete
265     variable query withValues (
266       | query.login_query | query.password_query | query.admin_email +|
267       | "' OR 1=1--" | "anything" | "admin@juice-sh.op" +|
268     )
269
270       step s1:System_Execute:OpenBrowser ["The system opens the browser in https:juice-shop.
              herokuapp.com/#/"]
271     step s2:Actor_CallSystem:Click element('Dismiss') ["The Visitor clicks the 'Dismiss'
            button"]
272     step s3:Actor_CallSystem:Click element('Account')[""]
273       step s4:Actor_CallSystem:Click element('SignIn') ["The Visitor clicks on the 'SignIn'
              button"]
274     step s5:Actor_PrepareData:PostData readFrom query.login_query ["The Cracker writes the
            query"]
275     step s6:Actor_PrepareData:PostData readFrom query.password_query ["The Cracker writes
            random data"]
276     step s7:Actor_CallSystem:Click element('Confirm_SignIn') ["The Visitor clicks on the '
            Sign In' button"]
277     step s8:Actor_CallSystem:Click element('My_Account') ["The Visitor clicks on the 'My
            Account' button"]
278     step s9:System_Execute:Check elementOnScreen(text query.admin_email) ["'MyAccount'
            appears with the user name"]
279   ]
280 ]*/
```

Listing A.1: RSL specification to the three Use Cases on Juice Shop

```
1 {
2   "sleep": "2s",
3   "selectors": [
4     {
5       "id": "Account",
```

```
      "selector": "(//SPAN)[20]"
    },
    {
      "id": "password_confirmation",
      "selector": "(//INPUT)[4]"
    },
    {
      "id": "t_uc_3_NewAddress",
      "selector": "(//BUTTON)[7]"
    },
    {
      "id": "t_uc_2_email",
      "selector": "(//INPUT)[2]"
    },
    {
      "id": "t_uc_3_Product",
      "selector": "(//BUTTON)[7]"
    },
    {
      "id": "t_uc_1_password",
      "selector": "(//INPUT)[3]"
    },
    {
      "id": "t_uc_3_ChooseAddress",
      "selector": "(//DIV)[67]"
    },
    {
      "id": "Purchase_PointAndClick",
      "selector": "(//H1)[1]"
    },
    {
      "id": "t_uc_3_state",
      "selector": "(//INPUT)[7]"
    },
    {
      "id": "t_uc_3_Continue2",
      "selector": "(//BUTTON)[8]"
    },
    {
      "id": "t_uc_3_Continue1",
      "selector": "(//BUTTON)[8]"
    },
    {
      "id": "SignIn_PointAndClick",
      "selector": "(//SPAN)[80]"
    },
    {
      "id": "t_uc_3_Continue3",
      "selector": "(//BUTTON)[16]"
    },
    {
      "id": "SecurityQuestion",
      "selector": "//div[@id='registration-form']/div[1]/mat-form-field[1]/div[1]/div[1]/div
          [3]"
```

```
    },
    {
      "id": "t_uc_3_country",
      "selector": "(//INPUT)[2]"
    },
    {
      "id": "t_uc_1_Dismiss",
      "selector": "(//BUTTON)[8]"
    },
    {
      "id": "My_Account",
      "selector": "(//BUTTON)[4]"
    },
    {
      "id": "t_uc_3_Checkout",
      "selector": "(//BUTTON)[10]"
    },
    {
      "id": "t_uc_3_Dismiss",
      "selector": "(//BUTTON)[9]"
    },
    {
      "id": "t_uc_2_Dismiss",
      "selector": "(//BUTTON)[10]"
    },
    {
      "id": "SecurityAnswer",
      "selector": "(//SPAN)[36]"
    },
    {
      "id": "t_uc_1_Confirm",
      "selector": "(//BUTTON)[6]"
    },
    {
      "id": "t_uc_3_city",
      "selector": "(//INPUT)[6]"
    },
    {
      "id": "t_uc_1_answer",
      "selector": "(//INPUT)[6]"
    },
    {
      "id": "t_uc_3_PlaceOrder",
      "selector": "(//BUTTON)[7]"
    },
    {
      "id": "SignUp_PointAndClick",
      "selector": "(//SPAN)[30]"
    },
    {
      "id": "t_uc_3_address",
      "selector": "(//TEXTAREA)[1]"
    },
    {
```

```
113       "id": "t_uc_1_email",
114       "selector": "(//INPUT)[2]"
115     },
116     {
117       "id": "t_uc_3_Basket",
118       "selector": "(//BUTTON)[5]"
119     },
120     {
121       "id": "t_uc_3_addressName",
122       "selector": "(//INPUT)[3]"
123     },
124     {
125       "id": "Confirm_SignIn",
126       "selector": "(//BUTTON)[7]"
127     },
128     {
129       "id": "t_uc_3_SubmitAddress",
130       "selector": "(//BUTTON)[8]"
131     },
132     {
133       "id": "t_uc_3_shippingMethod",
134       "selector": "(//DIV)[73]"
135     },
136     {
137       "id": "t_uc_3_mobileNumber",
138       "selector": "(//INPUT)[4]"
139     },
140     {
141       "id": "t_uc_3_ChooseCreditCard",
142       "selector": "(//DIV)[70]"
143     },
144     {
145       "id": "SignIn",
146       "selector": "css:#navbarLoginButton"
147     },
148     {
149       "id": "t_uc_2_password",
150       "selector": "(//INPUT)[3]"
151     },
152     {
153       "id": "t_uc_3_zipCode",
154       "selector": "(//INPUT)[5]"
155     }
156   ]
157 }
```

Listing A.2: Generated JSON

```
1 # Input data variables
2
3 t_uc_1_email1 = "fff@fff.com"
4 t_uc_1_password1 = "12345"
```

```
password_confirmation1 = "12345"
t_uc_1_answer1 = "abc123"
success_message1 = "Registration completed successfully. You can now log in."
t_uc_2_email1 = "ddd@ddd.com"
t_uc_2_accountEmail1 = "ddd@ddd.com"
t_uc_2_password1 = "12345"
t_uc_3_country1 = "Portugal"
t_uc_3_addressName1 = "FEUP"
t_uc_3_mobileNumber1 = "999 999 999"
t_uc_3_zipCode1 = "123456"
t_uc_3_address1 = "R. Dr. Roberto Frias"
t_uc_3_city1 = "Porto"
t_uc_3_state1 = "Porto"
Browser = "gc"

# Keyword Mapping variables

sleep = "2s"
Account = "(//SPAN)[20]"
password_confirmation = "(//INPUT)[4]"
t_uc_3_NewAddress = "(//BUTTON)[7]"
t_uc_2_email = "(//INPUT)[2]"
t_uc_3_Product = "(//BUTTON)[7]"
t_uc_1_password = "(//INPUT)[3]"
t_uc_3_ChooseAddress = "(//DIV)[67]"
Purchase_PointAndClick = "(//H1)[1]"
t_uc_3_state = "(//INPUT)[7]"
t_uc_3_Continue2 = "(//BUTTON)[8]"
t_uc_3_Continue1 = "(//BUTTON)[8]"
SignIn_PointAndClick = "(//SPAN)[80]"
t_uc_3_Continue3 = "(//BUTTON)[16]"
SecurityQuestion = "//div[@id='registration-form']/div[1]/mat-form-field[1]/div[1]/div[1]/div
    [3]"
t_uc_3_country = "(//INPUT)[2]"
t_uc_1_Dismiss = "(//BUTTON)[8]"
My_Account = "(//BUTTON)[4]"
t_uc_3_Checkout = "(//BUTTON)[10]"
t_uc_3_Dismiss = "(//BUTTON)[9]"
t_uc_2_Dismiss = "(//BUTTON)[10]"
SecurityAnswer = "(//SPAN)[36]"
t_uc_1_Confirm = "(//BUTTON)[6]"
t_uc_3_city = "(//INPUT)[6]"
t_uc_1_answer = "(//INPUT)[6]"
t_uc_3_PlaceOrder = "(//BUTTON)[7]"
SignUp_PointAndClick = "(//SPAN)[30]"
t_uc_3_address = "(//TEXTAREA)[1]"
t_uc_1_email = "(//INPUT)[2]"
t_uc_3_Basket = "(//BUTTON)[5]"
t_uc_3_addressName = "(//INPUT)[3]"
Confirm_SignIn = "(//BUTTON)[7]"
t_uc_3_SubmitAddress = "(//BUTTON)[8]"
t_uc_3_shippingMethod = "(//DIV)[73]"
t_uc_3_mobileNumber = "(//INPUT)[4]"
t_uc_3_ChooseCreditCard = "(//DIV)[70]"
```

```
SignIn = "css:#navbarLoginButton"
t_uc_2_password = "(//INPUT)[3]"
t_uc_3_zipCode = "(//INPUT)[5]"
```

Listing A.3: Python script containing all variables

```
*** Settings ***
Documentation               This is a basic test
Library                     Selenium2Library
Variables          JuiceShop.py

*** Test Cases ***


SignUp-Test_1
  [Documentation]     As a Visitor I want to sign up
  open browser  https://juice-shop.herokuapp.com/#/register  ${Browser}
  Sleep  2s
  Click element  ${t_uc_1_Dismiss}
  Sleep  2s
  input text   ${t_uc_1_email}   ${t_uc_1_email1}
  Sleep  2s
  input text   ${t_uc_1_password}   ${t_uc_1_password1}
  Sleep  2s
  input text   ${password_confirmation}   ${password_confirmation1}
  Sleep  2s
  Click element  ${SecurityQuestion}
  Sleep  2s
  Click element  ${SecurityAnswer}
  Sleep  2s
  input text   ${t_uc_1_answer}   ${t_uc_1_answer1}
  Sleep  2s
  Click element  ${t_uc_1_Confirm}
  Sleep  2s
  Page Should Contain Element ${SignUp_PointAndClick}  ${success_message1}


SignIn-Test_1
  [Documentation]     As a Visitor I want to sign in
  open browser  https://juice-shop.herokuapp.com/#/login  ${Browser}
  Sleep  2s
  Click element  ${t_uc_2_Dismiss}
  Sleep  2s
  input text   ${t_uc_2_email}   ${t_uc_2_email1}
  Sleep  2s
  input text   ${t_uc_2_password}   ${t_uc_2_password1}
  Sleep  2s
  Click element  ${Confirm_SignIn}
  Sleep  2s
  Click element  ${My_Account}
  Sleep  2s
  Page Should Contain Element ${SignIn_PointAndClick}  ${t_uc_2_accountEmail1}
```

```
47
48
49 Purchase-Test_1
50   [Documentation]     As a User I want to buy a product
51   open browser  https://juice-shop.herokuapp.com/#/  ${Browser}
52   Sleep  2s
53   Click element  ${t_uc_3_Dismiss}
54   Sleep  2s
55   Click element  ${Account}
56   Sleep  2s
57   Click element  ${SignIn}
58   Sleep  2s
59   input text   ${t_uc_2_email}   ${t_uc_2_email1}
60   Sleep  2s
61   input text   ${t_uc_2_password}   ${t_uc_2_password1}
62   Sleep  2s
63   Click element  ${Confirm_SignIn}
64   Sleep  2s
65   Click element  ${t_uc_3_Product}
66   Sleep  2s
67   Click element  ${t_uc_3_Basket}
68   Sleep  2s
69   Click element  ${t_uc_3_Checkout}
70   Sleep  2s
71   Click element  ${t_uc_3_NewAddress}
72   Sleep  2s
73   input text   ${t_uc_3_country}   ${t_uc_3_country1}
74   Sleep  2s
75   input text   ${t_uc_3_addressName}   ${t_uc_3_addressName1}
76   Sleep  2s
77   input text   ${t_uc_3_mobileNumber}   ${t_uc_3_mobileNumber1}
78   Sleep  2s
79   input text   ${t_uc_3_zipCode}   ${t_uc_3_zipCode1}
80   Sleep  2s
81   input text   ${t_uc_3_address}   ${t_uc_3_address1}
82   Sleep  2s
83   input text   ${t_uc_3_city}   ${t_uc_3_city1}
84   Sleep  2s
85   input text   ${t_uc_3_state}   ${t_uc_3_state1}
86   Sleep  2s
87   Click element  ${t_uc_3_SubmitAddress}
88   Sleep  2s
89   Click element  ${t_uc_3_ChooseAddress}
90   Sleep  2s
91   Click element  ${t_uc_3_Continue1}
92   Sleep  2s
93   Click element  ${t_uc_3_shippingMethod}
94   Sleep  2s
95   Click element  ${t_uc_3_Continue2}
96   Sleep  2s
97   Click element  ${t_uc_3_ChooseCreditCard}
98   Sleep  2s
99   Click element  ${t_uc_3_Continue3}
100  Sleep  2s
```

```
101  Click element  ${t_uc_3_PlaceOrder}
102  Sleep  2s
103  Page Should Contain Element ${Purchase_PointAndClick}  Thank you for your purchase!
```

Listing A.4: Robot Framework file containing the result script

References

[1] Sérgio Almeida, Ana C.R. Paiva, and André Restivo. Mutation-Based Web Test Case Generation. *Communications in Computer and Information Science*, 1010:339–346, 2019.

[2] Mahmoud Anouti. https://dzone.com/articles/guide-to-behavior-driven-development-in-java. https://dzone.com/articles/guide-to-behavior-driven-development-in-java, 2019.

[3] Ahlam Ansari, Mirza Baig Shagufta, Ansari Sadaf Fatima, and Shaikh Tehreem. Constructing Test cases using Natural Language Processing. *Proceedings of the 3rd IEEE International Conference on Advances in Electrical and Electronics, Information, Communication and Bio-Informatics, AEEICB 2017*, pages 95–99, 2017.

[4] Paulo J.M. Araújo and Ana C.R. Paiva. Pattern based web security testing. *MODELSWARD 2018 - Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, 2018-Janua(Modelsward):472–479, 2018.

[5] Paolo Arcaini and Elvinia Riccobene. Automatic refinement of ASM abstract test cases. *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2019*, pages 1–10, 2019.

[6] Alberto Rodrigues Da Silva. Linguistic patterns and linguistic styles for requirements specification (I): An application case with the rigorous rsl/business-level language. *ACM International Conference Proceeding Series*, Part F1320(I):1–27, 2017.

[7] Alberto Rodrigues da Silva, Ana C.R. Paiva, and Valter Emanuel R. da Silva. Towards a test specification language for information systems: Focus on data entity and state machine tests. *MODELSWARD 2018 - Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, 2018-Janua(Modelsward):213–224, 2018.

[8] Alberto Rodrigues da Silva, Ana C.R. Paiva, and Valter E.R. da Silva. A test specification language for information systems based on data entities, use cases and state machines. *Communications in Computer and Information Science*, 991:455–474, 2019.

[9] Valter E.R. da Silva. Model-Based Testing: From Requirements to Tests. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 2017.

[10] David De Almeida Ferreira and Alberto Rodrigues Da Silva. RSLingo: An information extraction approach toward formal requirements specifications. *2012 2nd IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012 - Proceedings*, pages 39–48, 2012.

[11] Eclipse Foundation. Eclipse Theia IDE. https://theia-ide.org/docs/, 2020.

[12] Eclipse Foundation. Xtext - Web Editor Support. https://www.eclipse.org/Xtext/documentation/330_web_support.html, 2020.

[13] EcuRed. Selenium IDE. https://medium.com/gangboard/selenium-ide-fd55b857e0ac, 2012.

[14] Eclipse Foundation. xtext-web source code. https://github.com/eclipse/xtext-web, 2008.

[15] Selenium Framework. Automation Practice | About Us. http://automationpractice.com/index.php?id_cms=4&controller=cms.

[16] Selenium Framework. Selenium Framework | Docs. https://www.selenium.dev/documentation/en/, 2020.

[17] Google. Google Chrome Extensions - Overview. https://developer.chrome.com/extensions, 2012.

[18] Gradle. Gradle Docs. https://docs.gradle.org/current/userguide/what_is_gradle.html, 2020.

[19] Maria Fernanda Granda. An experiment design for validating a test case generation strategy from requirements models. *2014 IEEE 4th International Workshop on Empirical Requirements Engineering, EmpiRE 2014 - Proceedings*, pages 44–47, 2014.

[20] Javier Gutiérrez, Gustavo Aragón, Manuel Mejías, Francisco Jose Domínguez Mayo, and Carmen M. Ruiz Cutilla. Automatic test case generation from functional requirements in NDT. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7703 LNCS:176–185, 2012.

[21] Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. *Proceedings - AGILE Conference, 2006*, 2006:270–275, 2006.

[22] B. Kimminich. OWASP Juice Shop. https://pwning.owasp-juice.shop/, 2020.

[23] Nora Koch, Matthias Pigerl, Gefei Zhang, and Tatiana Morozova. Patterns for the model-based development of RIAs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5648 LNCS(Ist 016004):283–291, 2009.

[24] Daniel Maciel, Ana C.R. Paiva, and Alberto Rodrigues Da Silva. From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach. *ENASE 2019 - Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 265–272, 2019.

[25] P. Mani and M. Prasanna. Validation of automated test cases with specification path. *Journal of Statistics and Management Systems*, 20(4):535–542, 2017.

[26] Jiri Matula, Frantisek Hunka, Jaroslav Zacek, and Czech Republic. Context Definition for Bdd Scenarios Upon. 2(3):58–63, 2016.

[27] João Miranda, Ana C R Paiva, and Alberto Rodrigues da Silva. Preliminary Experiences in Requirements-based Security Testing. 2:358–366, 2020.

[28] Rodrigo M.L.M. Moreira, Ana Cristina Paiva, Miguel Nabuco, and Atif Memon. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Software Testing Verification and Reliability*, 27(3), 2017.

[29] Inês Coimbra Morgado and Ana C.R. Paiva. Mobile GUI testing. *Software Quality Journal*, 26(4):1553–1570, 2018.

[30] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean Marc Jézéquel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.

[31] Ana C.R. Paiva, Daniel Maciel, and Alberto Rodrigues da Silva. From Requirements to Automated Acceptance Tests with the RSL Language. *Communications in Computer and Information Science*, 1172 CCIS:39–57, 2020.

[32] Ana C.R. Paiva, André Restivo, and Sérgio Almeida. Test case generation based on mutations over user execution traces. *Software Quality Journal*, pages 1173–1186, 2020.

[33] Robot Framework. Robot Framework. https://robotframework.org/#documentation, 2008.

[34] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. On the evolution of keyword-driven test suites. *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pages 335–345, 2019.

[35] Adam Sadovsky. XPath Helper - Browser Extension. https://chrome.google.com/webstore/detail/xpath-helper/hgimnogjllphhhkhlmebbmlgjoejdpjl, 2015.

[36] Alberto Rodrigues Silva. Rigorous Requirements Specification with the RSL Language: Focus on Uses Cases. *Proceedings of ISD' 2019, AIS*, (August), 2019.

[37] Pedro Silva, Ana C.R. Paiva, Andre Restivo, and Jorge Esparteiro Garcia. Automatic test case generation from usage information. *Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018*, pages 268–271, 2018.

[38] Kapil Singi, Dipin Era, and Vikrant Kaulgud. Model-based approach for automated test case generation from visual requirement specifications. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2015 - Proceedings*, pages 1–6, 2015.

[39] John Sonmez. How to Create a Chrome Extension in 10 Minutes Flat. https://www.sitepoint.com/create-chrome-extension-10-minutes-flat/, 2015.

[40] Testbytes. Selenium IDE Tutorial For Beginners. https://medium.com/@testbytessoftware/selenium-ide-tutorial-for-beginners-testbytes-8e917bfdc8e8, 2019.

[41] Testleaf. RUTO. https://sites.google.com/testleaf.com/ruto/, 2020.

[42] Typefox. Theia Xtext Sprotty Example. https://github.com/TypeFox/theia-xtext-sprotty-example.

[43] TypeFox. DSLs in the Cloud - with Eclipse Technologies, 2018.

[44] Webscraper.io. Web Scraper - Free Web Scraping. https://webscraper.io/, 2020.