FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Model-based testing: From requirements to tests

Daniel Ademar Magalhães Maciel

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Professor Ana Cristina Ramada Paiva

Second Supervisor: Professor Alberto Manuel Rodrigues da Silva

February 21, 2019

Model-based testing: From requirements to tests

Daniel Ademar Magalhães Maciel

Mestrado Integrado em Engenharia Informática e Computação

February 21, 2019

Abstract

Software testing is increasingly valued as it promotes the quality of the software by checking that it meets the expected requirements. Frequently, software testing tends to be neglected at the beginning of the projects, only performed on the late stage. However, it is possible to benefit from combining testing with requirement specification activities.

On one hand, acceptance tests specification will require less manual effort since they are defined or generated automatically from the requirements specification. On the other hand, the requirements specification itself will end up having higher quality due to the use of a more structured language, reducing typical problems such as ambiguity, inconsistency and incorrectness of requirements.

In this research we propose an approach that promotes the practice of tests specification since the very beginning of projects, and its integration with the requirements specification itself. This approach is conducted by model-driven techniques that contribute to maintain the requirements and tests alignment, namely alignment between requirements (defined as use cases), test cases, and low-level automated test scripts.

To show the applicability of this approach we integrate two complementary languages: (i) the RSL (Requirements Specification Language) that is particularly designed to support both requirements and tests specification in a rigorous and consistent way; and (ii) the Robot language, which is a low-level keyword-based language for the specification of test scripts. In addition, the proposed approach includes model-to-model transformation techniques, such as requirements to test cases transformation, and test cases into test scripts transformation. Finally, these test scripts are executed by the Robot test automation framework.

The approach developed was applied in a fictitious Web Store that serves as an illustrative example. With this example is shown how this approach can cover the project life-cycle from the requirements to tests.

Resumo

O Teste de Software é cada vez mais valorizado, uma vez que promove a qualidade do Software ao verificar se este coincide com os requisitos especificados. Normalmente, a prática de testes tende a ser negligenciada no início dos projectos, sendo só iniciada em fases mais avançadas dos projectos. No entanto, quando se combina as actividades de testes e especificação de requisitos é possível adquirir mais benefícios.

Por um lado, as especificações de testes de aceitação vão exigir menor esforço manual, uma vez que estes serão definidos ou gerados automaticamente a partir das especificações de requisitos. Por outro lado, a própria especificação de requisitos vai acabar por ter maior qualidade, devido ao uso de uma linguagem mais estruturada capaz de reduzir problemas tipicamentos associados aos requisitos, como a ambiguidade e inconsistência.

Nesta investigação, propomos uma abordagem que promove a prática de especificação de testes e a sua integração com a especificação de requisitos desde o início do projecto. Nesta abordagem são utilizadas técnicas baseadas em modelos, que contribuem para manter o alinhamento entre os requisitos e os testes, isto é, o alinhamento entre requisitos (definidos como casos de uso), casos de teste e scripts de testes de baixo nível automatizados.

Para mostrar a aplicabilidade desta abordagem, integramos duas linguagens complementares: (i) a RSL (Requirements Specification Language), que é particularmente projetada para suportar os requisitos e a especificação de testes de uma maneira rigorosa e consistente; e (ii) a linguagem Robot, que é um idioma de baixo nível baseado em palavras-chave para a especificação de scripts de teste. Além disso, a abordagem proposta inclui técnicas de transformação de modelo para modelo, nomeadamente transformações de requisitos em casos de teste e transformações desses casos de teste em scripts de teste. Por fim, esses scripts de teste são executados pela estrutura de automação de teste do Robot.

A abordagem desenvolvida foi aplicada numa loja virtual fictícia com o intuito de servir como exemplo ilustrativo. Com isto, é mostrado como esta abordagem pode cobrir o ciclo de vida do projeto, desde os requisitos até os testes.

Acknowledgements

In the face of my academic career and the last stage of this journey, I would like to thank the Professor Ana Cristina Ramada Paiva for the skill and wisdom with which she guided the development of the presented monograph and the teachings given to me during the degree of Masters of Informatics Engineering and Computing in this noble faculty, and all the professors, teachers and engineers who have made a mark in this path, assuring quality in education and in the transmission of human values. I would also like to thank Professor Alberto Manuel Rodrigues da Silva, who, despite being from another University, demonstrated total availability, commitment and professionalism and whose guidance and support were fundamental for the development of this dissertation.

I want to give recognition to my family for the long years of perseverance and struggle at my side, for encouraging my dedication and effort and for sharing the same dream: to my sister Catarina for sharing with me aspirations and challenges throughout her life, for not having stopped trusting me and have been a safe haven and mutual example of dedication, my mother for the unconditional support, love and for the undeniable effort to achieve this feat, Luís Martins, for his support, motivation, positivism and availability, my godmother, for always having a word of comfort, for her constant concern and confidence in me, my grandmother Gloria, for being a great source of inspiration and protection, my father, my grandmother Conceição, my uncles, and the rest of the family, who always believed in the completion of this degree.

I also want to highlight my classmates, who have shared years of tenacity and dedication to a single objective with a spirit of unity and camaraderie, especially Adriana Teles, Bruno Santos, Bruno Oliveira, César Silva, José Pedro Teles and João Pedro Mendonça.

I also want to thank my friends, especially Rita Azevedo, Helena Moreira, and Ana Isabel Moreira, for the constant expression of friendship, for being a support in times of need, and for always supporting me with sweetness and protection.

Daniel Maciel

"Learning never exhausts the mind"

Leonardo Da Vinci

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

AST	Abstract Syntax Tree
ATDD	Acceptance Test-driven Development
AUT	Application Under Test
BRD	Business Requirements Document
DOM	Document Object Model
DSL	Domain Specific Language
EUC	Essential Use Case
FEUP	Faculdade de Engenharia da Universidade do Porto
GPL	General Purpose Languages
GUI	Graphical User Interface
IST	Instituto Superior Técnico da Universidade de Lisboa
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MBT	Model-based Testing
NLP	Natural Language Processing
PBGT	Pattern Based GUI Testing
RE	Requirements Engineering
RSL	Requirements Specification Language
SRS	Software Requirements Specification
UI	User Interface

Chapter 1

Introduction

This research work addresses the field of Software Engineering, specifically in the area of Software Testing and Requirements Engineering. This chapter provides an overview of the research work by presenting the context, motivation and objectives of the dissertation. In addition it also presents the general structure of the document.

1.1 Context

Software systems are constantly evolving becoming more complex, which increases the need for efficient and regular testing activity to ensure quality and increase the product confidence. Software systems' quality is usually evaluated by the software product's ability to meet the implicit and explicit customer needs. For this purpose, it is important that customers and developers achieve a mutual understanding of the features of the software that will be developed.

Requirements Engineering (RE) is a socio-technical discipline that intends to provide a shared vision and understanding of systems among the involved stakeholders and throughout its life-cycle. The system requirements specification (SRS) is an important document that helps to structure the system's concerns from an RE perspective and offers several benefits, already reported in literature [Coc01, Kov98, RR06, Wit07], such as the establishment of an agreement between consumers and developers, support validation and verification of the project scope, and support future system maintenance activities. The problem is that the manual effort required to produce requirements specifications is high and, given the large amount of information to be considered, this activity suffers from problems, such as, incorrectness, inconsistency, incompleteness, and ambiguity [Kov98, RR06, Poh10].

ITLingo is a long term initiative with the goal to research, develop and apply rigorous specification languages in the IT domain, namely Requirements Engineering, Testing Engineering and Project Management [Sil18]. ITLingo adopts a linguistic approach to improve the rigorous of technical documentation (e.g., SRS, test case specification, project plans), and as a consequence to promote productivity through re-usability and model transformations, as well as promote quality through semi-automatic validation techniques.

RSL (Requirements Specification Language) is a controlled natural language integrated in ITLingo which helps the production of requirements specifications in a systematic, rigorous and consistent way [Sil17]. RSL includes a rich set of constructs logically arranged into views according to RE-specific concerns that exist at different abstraction levels, such as business, application, software or even hardware levels.

Software testing can be used to track and evaluate the software development process by measuring the number of tests that pass or fail and by performing continuous regression testing that allows to maintain the product quality alerting developers of possible defects as soon as the code is changed.

In the last decades new approaches have been adopted, especially in agile development, which aims to automate testing activities. The automation of tests has proven to be an initiative that promotes the ease and speed of the process [HH08].

The Model-based testing (MBT) is a software testing approach, that generates test cases from abstract representations of the system, named models, either graphical (e.g., Workflow models [BM17], PBGT [MPNM17, MP14]) or textual (e.g., requirements documents in an intermediate format)[Pai07].

> “Good requirements engineering produces better tests; good test analysis produces better requirements.” [Gra02]

Being Requirements Engineering and Software Testing activities with a great relation and synergy, linking these activities brings benefits in both directions while helps to save time and money. Within the different types of tests, the acceptance tests are those that have a greater relationship with the requirements [HCG16].

However, even though it is considered good practice to beginning testing at the start of the project, when requirements are raised, this is not always the case due to the high manual effort and the separation of the requirements phase from the testing phase in traditional processes.

1.2 Motivation and Objectives

In automated testing, it is used software to perform or support test activities, which test the application under test (AUT) behavior, without human intervention. When planned and implemented properly, automated testing can yield various benefits over manual testing, such as repeatability and reduced test costs and efforts [GE17]. With the test automation, in particular, through the Model-based Testing approach, it is possible to generate test cases from SRS specified in RSL, which, in addition to reducing the manual effort, also ensures higher quality requirements.

Among different types of tests, the acceptance tests [Chr08] are those that have a greater relationship with the requirements. They are used to test with respect to the user needs, requirements and business processes, and are conducted to determine whether or not a system satisfies the acceptance criteria and to enable users, customers or other authorized entities to determine whether or not shall accept the system [SQ15]. In order to improve the acceptance testing and requirements specification activities, it may be advantageous to perform these activities in parallel which, in addition to increasing the quality of the requirements, also allows to reduce the time and resources spent with them. Even though, starting the testing activities at the beginning of the project when the requirements are elicited is considered a good practice, this is not always the case because requirements elicitation and testing are separate in traditional development processes.

This research work seeks to adopt the Model-based testing approach with the support of the RSL language that, based on the requirements specification, automatically generates part of the acceptance tests, as well as the scripts that can be executed when the application is already available.

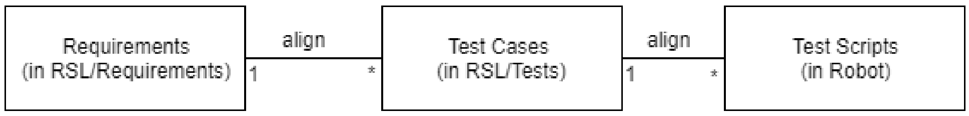


Figure 1.1: Approach terminologies

Following the Figure 1.1, this approach uses RSL *Requirements* specifications produced through a set of constructs provided by the language according to different concerns. Then, each *Requirement* is aligned with RSL *Test Cases* specifications. From the RSL *Test Cases* specifications, it is possible to align and generate *Test scripts* that can be executed automatically by the Robot[1] test automation tool over the AUT.

The objective of this research is to support the starting of the practice of tests at the beginning of a project when the requirements are specified. For this, the approach studied aims to:

- Investigate test automation mechanisms from abstract specifications.
- Transform RSL requirements specifications into high-level tests.
- Perform end-to-end integration from requirement specification to acceptance test execution.
- Automation of acceptance tests for interactive applications.

It is expected that the framework developed during this research would be able to generate and execute acceptance tests and, consequently, to encourage the beginning of the test practice at the beginning of the project.

[1]http://robotframework.org/

1.3 Structure of Dissertation

In addition to the current chapter, Introduction, this dissertation has four more chapters.

In Chapter 2 is presented the State of Art where the fundamental concepts of the scope of the dissertation are analyzed and studied, as well as related works capable of generating test cases from a given format. Finally, a comparison is made of the main tools available in the market dedicated to the automation of tests.

Chapter 3 describes the context and grammar of the RSL and the most relevant constructs to the tests specification.

Chapter 4 describes in detail the approach studied to solve the problem of separation of the requirement elicitation phase with the testing phase.

Chapter 5 presents an illustrative example where the technique is applied.

Chapter 6 concludes the dissertation with an overview of the results obtained. In addition, we also present suggestions for future work that could help to improve our work.

Chapter 2

State of Art

This chapter introduces the state of art and reviews the existing literature. First, it presents the fundamental concepts belonging to the background of the research work. In this way, aspects related to RE, Software Testing and Domain Specific Language (DSL) are addressed, which bring value to the study for its importance and need to have this knowledge.

Second, it analyses tools and approaches similar to the proposed work. For instance, approaches with test cases generation, test execution and/or requirements quality improvement purposes. Third, it presents a study of automation tools for test execution so that it's possible to select a tool to be integrated into the solution.

2.1 Requirements Engineering

According to the certified glossary of Requirements Engineering [Gli14], requirements engineering (RE) is a systematic approach to the management and specification of the conditions necessary to solve a problem or to achieve an objective. The goals of this area are to reach consensus among stakeholders, understand and document their needs and minimize the risks of deliveries that do not meet those needs.

RE is a key part of software development since there are usually a large number of stakeholder specific needs to complete the idea. With this in mind, it is necessary to gather information to discover what the software will be able to do and who the target audience will be. Most requirements processes, in addition to analyzing stakeholder needs, also have to take into account the requirements of the development environment.

> "Design requirements represent a crossroads where several research, business, engineering, and artistic communities converge." [HBL09, chap. Introduction]

The quality of RE is an important point to consider. Deficient interactions between engineers and clients can cause requirements to be incomplete or even require changes in requirements throughout the project, affecting the product development process and causing missed deadlines, and therefore increasing the cost of product development. In fact, a large number of software life cycles fail because of a bad requirements engineering. According to Lindquist [Lin05], 71% of failures in software projects are associated with poor requirements management.

One of the ways to improve requirements quality is starting the testing activity during the requirements phase. In fact, involving testers during the requirements analysis is one of the best ways to ensure good requirements [Gra02].

According to the study done in [UKKD08], there are some good practices that can be applied to make the two activities closer. For instance, involving testers in requirements reviews, which result in higher requirements quality and improved testability, and establishing traceability between tests and requirements, which improve the test coverage and the efficiency of change management when there are requirements changes.

2.1.1 Types of Requirements

Software requirements can be categorized into different types of classification. For example, in the software area, there are three levels of requirements [WB13]: Business requirements, User requirements, and System Requirements which are categorized as either Functional or Nonfunctional [Shu16].

- **Business Requirements:** Business requirements refer to the high-level goal for which the organization wants to develop the product or the reason why the customer is looking for it. Usually this type of requirement is included in a Business Requirements Document (BRD) along with the problem, the project vision, project constraints, business objectives, project scope, business process analysis, stakeholder analysis and IT service impact [Shu16].

- **User Requirements:** User requirements are the goals or tasks that a given group of users should be able to do on the system, i.e. describe what the user will be able to do with the system. The user requirements also includes descriptions of product attributes or characteristics that are important to user satisfaction. Usually, this type of requirements is represented in the form of Use Cases or User Stories [WB13].

- **Functional Requirements:** Functional requirements specify the behaviors that the system will take when subject to certain conditions. This type of requirements describes what developers must implement to enable users to accomplish a given task (user requirements) while satisfying the business requirements. These functional requirements are documented in the Software Requirements Specification (SRS) that describes the expected behavior of the system. This document is used for different purposes, for example to support development, testing, quality assurance and even project management [WB13].

- **Nonfunctional Requirements:** Nonfunctional requirements are a type of requirement that specifies the type of criteria that will be used to judge the quality of a system as a whole, rather than just some kind of specific function. Nonfunctional requirements are also called as quality attributes, supplemental requirements or service level requirements. Non-functional requirements may include availability, business continuity, portability, reliability, testability, efficiency and modifiability, compliance, interoperability, maintainability, performance, or usability [Shu16]. SRS also contains non-functional requirements.

2.1.2 Requirements Engineering Process

The RE process, involves several activities are carried out, namely Elicitation, Analysis and Negotiation, Specification and Validation and Verification [Shu16]. Figure 2.1 shows a RE systematic process.

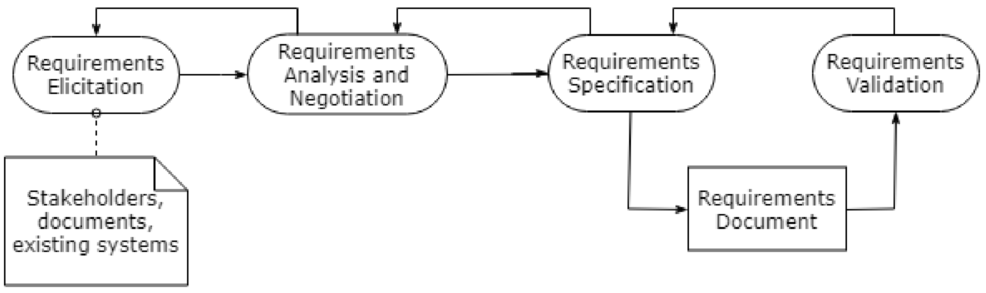

Requirements Elicitation
Requirements Analysis and Negotiation
Requirements Specification
Requirements Validation
Stakeholders, documents, existing systems
Requirements Document

Figure 2.1: Requirements Engineering process

Elicitation is the first activity of the process where the team must determine which organizational or client needs are addressed by the artifacts and also allows knowledge to be gathered from the application domain. According to [PR15], there are three sources of knowledge, these being the stakeholders, who are people or organizations that directly or indirectly influence the requirements of the system, the documents that contain important information and that are able to provide requirements and the systems in operation such as predecessor systems or even competing systems. There are a number of techniques that support the collection of such knowledge, within which Introspection, Interviews, Group-focused Discussion, Direct Observation of Business Processes and Prototyping. The way the discovery process is structured affects both the quality and the quantity of the requirements, since the combination of techniques allows multiple retrospectives to be adopted in the application domain.

In the requirements **analysis and negotiation** activities are identified the conflicts, analyzed the causes, resolved the conflicts and documented the solutions found. Conflicts can arise during any requirement engineering activity and are not always obvious. The basic strategies for resolving requirements conflicts go through negotiation where proposals and counter-proposals are made until consent is reached, creative solutions where the conflicting viewpoints are discarded and a new solution is found, or lastly, the highest authority takes the decision to resolve the conflict.

The **Specification** activity is intended to support the interpretation and understanding of all project stakeholders based on what the artifact must contain while establishing a sufficient technical basis for development. The Specification marks the transition point where the needs stated by the stakeholders will be extended with the functional and technical implications that arise from them. There are some fundamental concepts for the discussion of the specification of the requirements, being the Abstraction, where it is sought to extract the necessary details for the development, the Decomposition in which the system is divided into components, the Traceability that allows the organization of the requirements and complements the decomposition and the Natural Language and Modeling where the requirements/specifications are represented in a more symbolic way.

Validation verifies whether or not requirements are effectively specified and if is the right product that is being developed, i.e., it ensures that the requirements accurately reflect the intentions of the stakeholders. Particularly in agile approaches, the use of acceptance tests is encouraged to validate requirements and verify if the system reacts as expected [LQ10].

2.1.3 Quality of Requirements Specifications

SRS is used in different phases of the project to help stakeholders understand the vision of the system, to facilitate communication, to manage the project and to support the system development process. Normally, in these documents is used the natural language because it is a flexible and universal language. However, this language contains some characteristics that can cause some problems related to the requirements quality such as inconsistency, incompleteness, and ambiguousness [Poh10].

According to [Sil14] and [IEE98], to achieve quality, a SRS shall be:

- **Complete:** To be complete, the SRS should include everything that the system is supposed to do, the syntactic structure filled in and no section or item to be determined. In addition, to achieve completeness, reviews by the customer or users are required.
- **Consistent:** A SRS is consistent when no requirement is in conflict. Inconsistencies may appear when there are changes in requirements and there are no reviews of the requirements that are related to those changed requirements.
- **Unambiguous:** In order for SRS not to be ambiguous, all the requirements can only have one possible interpretation. Ambiguity can be introduced unintentionally by the simple use of natural language, since a word can have different meanings. To minimize this situation it is recommended to use formal or semi-formal specification languages.

2.2 Software Testing

Software products are prone to failure. Since it is usually impossible to prove the absence of bugs in a given program, the primary goal of software testing is to find failures as early as possible so that they can be fixed [Pat01].

With the evolution of the software systems complexity, it becomes increasingly difficult to ensure the quality of the products. It is therefore important to maintain a regular testing practice in order to increase product confidence and at the same time ensure that the actual behavior matches the expected behavior.

As refered by ISTQB [IST], Software Testing is the process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

2.2.1 Test Levels

Tests can be defined or derived from different resources such as requirements and related specifications, design artifacts or by the source code itself. Throughout software development, different levels of testing accompany each developmental activity. For example, ISTQB defined the following the test levels [IST]:

- **Unit Testing:** The testing of individual hardware or software components.
- **Integration Testing:** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.
- **System Testing:** Testing an integrated system to verify that it meets specified requirements.
- **Acceptance Testing:** Testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customer or other authorized entity to determine whether or not to accept the system.

2.2.1.1 Acceptance Tests

In the requirements phase of the software development, the customer needs are elicited. Acceptance tests are then performed to verify that the completed software meets those needs, i.e., these tests prove that the product does what the user wants. This type of tests should involve users and other stakeholders with a strong domain knowledge [AO16].

There are different types of acceptance tests [IST]:

- **User Acceptance Tests:** Tests that focus on user requirements, and business processes.
- **Operational Acceptance Testing:** Tests focus on operational aspects, e.g., the recoverability, installability, resource-behavior and technical compliance.

- **Alpha Tests:** Tests performed by potential users or independent teams at the developer's site, but outside the development organization. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing before the software goes to beta testing [Int10].
- **Beta Tests:** Tests performed by potential or existing users at an external site not otherwise involved with the developers in order to determine if the system meets user needs and fits into the business process. Beta testing is often employed as a form of external acceptance testing for commercial off-the-shelf software in order to acquire feedback from the market [Int10].
- **Site Acceptance Tests:** Tests performed by the users/customers at their site, to determine if the system meets their needs and fits into the business process, where both the hardware and the software are included.

In acceptance testing it is necessary to define criteria, i.e., acceptance criteria, that determine what component or system must be satisfied to be accepted. For instance, these criteria can be defined as [NT11]: Functional Correctness and Completeness, Accuracy, Data Integrity, Data Conversion, Backup and Recovery, Competitive Edge, Usability, Performance, Start-up Time, Stress, Reliability and Availability, Maintainability and Serviceability, Robustness, Timeliness, Confidentiality and Availability, Compliance, Installability and Upgradability, Scalability and Documentation.

2.2.2 Model-based Testing

Despite the importance of the testing process, the greater the complexity and dimension of the system, the greater are the costs and the time required to manually test the system. For that reason, the use of approaches capable of automating some of the test processes and at the same time maintaining or even improving quality is increasingly valued [MP14].

Model-Based Testing (MBT) refers to a software engineering process that studies, constructs, analyzes and applies well-defined models to support the various activities related to the tests.

> "MBT relates to a process of test generation from models of/related to a system under test (SUT) by applying a number of sophisticated methods. The basic idea of MBT is that instead of creating test cases manually, a selected algorithm is generating them automatically from a model." [ZSM17, chap. Introduction]

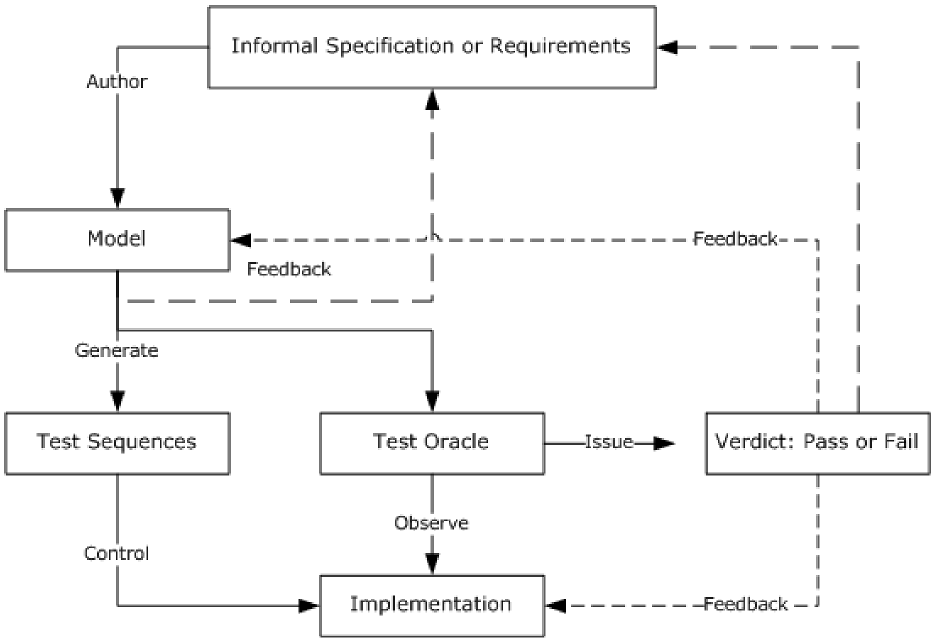


Figure 2.2: Model-based testing workflow (extracted from [MB-18])

The simplified MBT process is outlined in the Figure 2.2. Typically, specifications and requirements create the model, resulting in feedback on the initial specifications, since the design process requires that models question themselves in order to detect lack of information or to seek clarity. Then the test suites are generated automatically through the model, containing the Test Sequences and Test Oracle. The test sequences will control the SUT, driving it to different conditions that can be tested in accordance with the model. The oracle will observe the progress of the implementation and decide whether to pass or fail. This verdict will provide information about all artifacts. The failure indicates that the behaviour of the system under test does not meet the model's predictions, which usually indicates that there has been a failure in implementation, model creation or even requirements.

As described in Model-Based Tester Syllabus, establishing traceability between requirements, MBT models, and generated tests is a good practice. This link between requirements and model elements is associated with the following benefits [IST15]:

- Reviewing MBT models is simpler.
- The tests generated by the approach can be automatically associated with the requirements.
- It is possible to generate tests based on a selection of requirements and also prioritize them according to the priority of the requirements selected.
- It allows to measure the coverage of the requirements through the tests generated.
- It allows all stakeholders to analyze the impact of requirements changes and determine the scope required for regression testing.
- Test case generators can generate traceability documents automatically.

2.2.2.1 PBGT

Pattern Based GUI Testing (PBGT) is a model-based methodology for systematizing and automate the testing process in graphical interfaces [MP14]. This approach is supported by a tool, with the same name, which provides an integrated modeling and testing environment that allows the design of models based on UI Test Pattern, using the PARADIGM, a DSL developed specifically for this methodology. UI Test Pattern are elements included in PARADIGM and from which it is possible to create models that describe the objectives of GUI tests in a high-level abstraction [MP14].

The support tool, integrated as a plug-in in Eclipse, is divided into 5 main components [MP14]:

- **PARADIGM:** a DSL for building GUI test models based on UI Test Patterns;
- **PARADIGM-RE:** a reverse engineering component;
- **PARADIGM-TG:** an automated test case generation component;
- **PARADIGM-TE:** a test case execution component;
- **PARADIGM-ME:** a modeling environment that supports the building and configuration of test models;

In addition to these, an extra component, PARADIGM-COV, has been developed to analyze the coverage of the generated tests [PV17].

The process is divided into 6 steps [MP14]: modeling, configuration, test case generation, test case execution, results analysis and model update. First, in the modeling phase that can be done either manually or automatically through the PARADIGM-RE, which obtains part of the model by applying reverse engineering in the existing SUT, implying in this way a PARADIGM model with the appropriate UI Test Patterns. Second, the software testers configure each UI Test Pattern with input data, preconditions, and verifications. Afterwards, the PARADIGM-TG generates the test cases considering the model and the configurations provided. Finally, in the PARADIGM-TE component, the generated tests are executed and an analysis on an execution report, produced by the tool, is performed. In case of need, the model can be updated through PARADIGM-ME.

2.2.3 GUI Testing

The interaction between the user and the software is done through the program's Graphical User Interface (GUI). GUIs are the established medium of interaction with computer systems and can be a crucial point in the users' decisions to use or not use the system [ACRPV05]. It is important to have GUI testing to ensure correct operation and to detect defects. However, conventional test methods do not cover the GUI.

> "GUI testing is extremely time-consuming and costly. Applications are becoming bigger and more complex and manual testing of GUIs is labor-intensive, frequently monotonous, and is becoming an even more difficult activity." [ACRPV05]

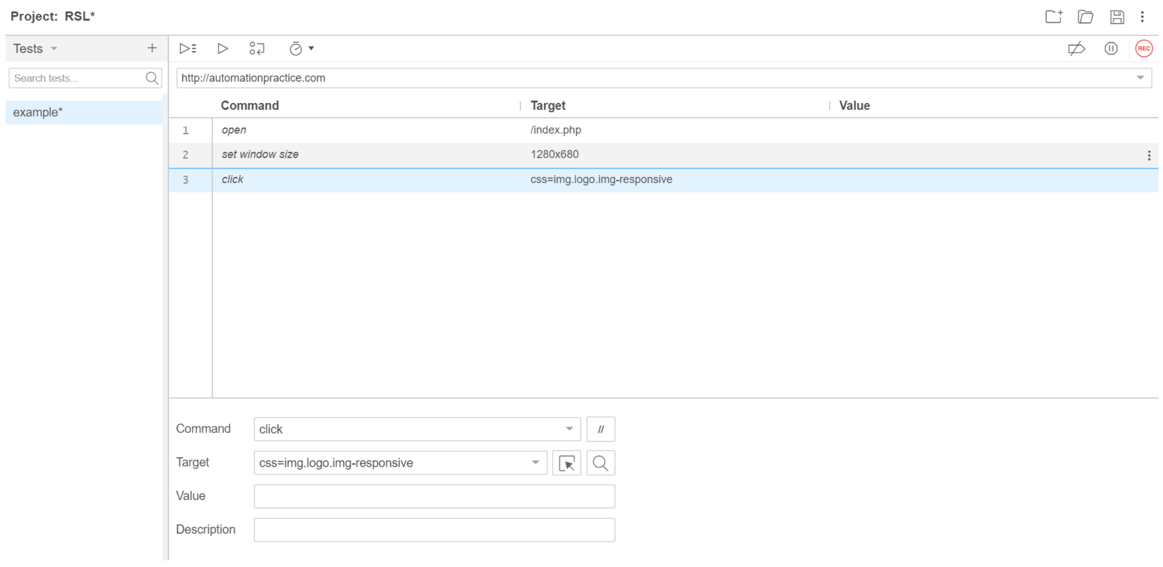
GUI Capture and Replay tools are developed and used to increase process automation and, consequently, decrease manual effort. By using this type of tools, testers can run the application and capture the interactions between the user and the application itself. From there a test script is written that contains all the user actions. In this way, the tool is able to replay the same interactions several times without the need for human intervention. However, these tools still require too much manual effort and postpone testing to the end of the development process, when the GUI is already constructed and functional. They are useful mainly for regression testing [PFV07].

In addition to Capture and Replay tools, there are still other approaches that reduce the manual work required to test an application through its GUI. Paiva et al. [PFV07] proposes a model pattern for GUIs and a GUI mapping tool that bridges the gap between a model written in a high-level modelling language and the simulation of user events and promotes a modelling pattern in which GUI components can be specified as reusable classes controlled by a window manager.

2.2.3.1 Selenium IDE

Selenium IDE is an integrated development environment that allows to capture actions, record them in test scripts, edit and debug tests. Records several locators for each element that interacts. If one fails during playback, others will be tried until one is successful. This IDE includes the entire Selenium core allowing to record and playback tests in the actual environment that they will run in. The minimum knowledge required to use this tool is the basic of HTML, DOMs and Javascript structures, not being necessary any prior programming knowledge.

Figure 2.3 shows the user interface of the Selenium IDE and the Listing 2.1 is an example of the generated script.


Project: RSL*

Tests

Search tests...

example*

http://automationpractice.com

	Command	Target	Value
1	open	/index.php	
2	set window size	1280x680	
3	click	css=img.logo.img-responsive	

Command	click
Target	css=img.logo.img-responsive
Value	
Description	

Figure 2.3: Selenium IDE

```
{
"id": "34a6a51b-64a4-4041-be0e-4f9605b17747",
  "version": "1.1",
  "name": "RSL",
  "url": "http://automationpractice.com",
  "tests": [{
    "id": "5da4fccf-dc3a-4f11-8f0c-d68762ee7451",
    "name": "example",
    "commands": [
    ...,{
      "id": "04ef3d9a-8571-4894-9d08-37061f84d5ab",
      "comment": "Home",
      "command": "click",
      "target": "css=img.logo.img-responsive",
      "targets": [
        ["css=img.logo.img-responsive", "css"],
        ["css=.logo", "css:finder"],
        ["xpath=//img[@alt='My Store']", "xpath:img"],
        ["xpath=//div[@id='header_logo']/a/img", "xpath:idRelative"],
        ["xpath=//div[3]/div/div/div/a/img", "xpath:position"]
      ],
      "value": ""
    }]
  }],
  ...
}
```

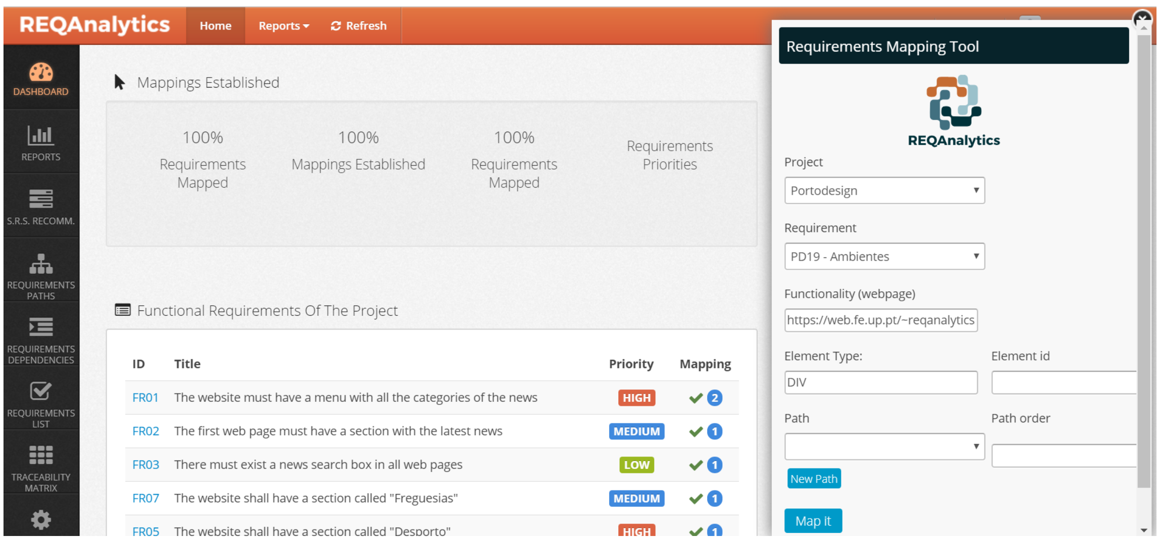
Listing 2.1: Excerpt of the generated Selenium script

2.2.3.2 REQAnalytics

REQAnalytics [GP16] (Figure 2.4) is a system that analyzes the usage and navigation made in the various pages of a web site and also generates recommendations to improve the quality of the requirements specification. This system is divided into four distinct phases [GP16]:

- mapping requirements with application features;
- web usage data collection;
- analysis between the data collected in previous phases;
- high level recommendation report generation;

In the first phase, the functional requirements of the website under analysis are mapped with the web pages and HTML elements present in the website. To establish the mapping between requirements and the web elements, the user must select a requirement in a check box (where are all the functional requirements previously imported through the XML document), and point/click on the page and/or HTML element that is related with this requirement [GP16].



Mappings Established

100% Requirements Mapped | 100% Mappings Established | 100% Requirements Mapped | Requirements Priorities

Functional Requirements Of The Project

ID	Title	Priority	Mapping
FR01	The website must have a menu with all the categories of the news	HIGH	✓ 2
FR02	The first web page must have a section with the latest news	MEDIUM	✓ 1
FR03	There must exist a news search box in all web pages	LOW	✓ 1
FR07	The website shall have a section called "Freguesias"	MEDIUM	✓ 1
FR05	The website shall have a section called "Desporto"	HIGH	✓ 1

Requirements Mapping Tool

REQAnalytics

Project: Portodesign

Requirement: PD19 - Ambientes

Functionality (webpage): https://web.fe.up.pt/~reqanalytics

Element Type: DIV

Element id:

Path:

Path order:

New Path

Map it

Figure 2.4: REQAnalytics User Interface

2.2.3.3 Web Scrapper

Web Scrapper[1] is a browser extension that allows extracting data from web applications (the term 'web scraping' itself is also used to define the data extraction of websites). Since some GUI testing approaches require the locators of the AUT elements, this extension becomes advantageous, since it allows the extraction of these locators too. The use of Web Scrapper is based on the creation of a sitemap, i.e., file used to provide information about the page and indicate the existing relationships. It associates a starting URL, which represents the page on which the data will be collected.

[1]https://www.webscraper.io/

Next, the locators, referred to as selectors in the context of the extension, are added. These locators are inserted into the sitemap through the point and click technique. When the sitemap is complete, it is possible to automatically extract the data from the web page.

The main purpose of this tool, i.e., web page data extraction, is outside of the intended GUI testing context. However, Web Scrapper allows the extraction of the sitemap with the locators in JSON code or as a CSV file. This extracted information turns out to be useful for some GUI testing approaches.

2.3 Domain Specific Language

Domain Specific Languages (DSL) are programming languages or specification languages developed and used to cover a specific domain problem. Unlike General Purpose Languages (GPL) (e.g. Java and C), DSLs are not meant to provide features that solve all kinds of problems. On the other hand, they allow to solve problems more easily and quickly. Some of the best known and used DSLs are SQL and HTML.

To develop a DSL, it is necessary to develop a program capable of reading text in this DSL, analyzing it, parsing it and eventually interpreting it and generating code in another language, depending on the purpose. When reading a program written in a programming language, the implementation has to ensure that the program matches the syntax of the language. For this, it is necessary to do a syntactic analysis, also known as parsing [Bet16]. There are already tools to parse, removing the dedicated effort to implement a parser. In addition, there are DSLs to specify language grammar. From the specification, the code for the lexer and the parser is automatically generated. Grammar is a set of rules that describe according to the syntax of the language, if the form of the elements is valid.

After syntactic analysis is done, it is also necessary to have a semantic analysis. Checking types, i.e. verifying that instances of variables only can be declared in the respective type, is part of the semantic analysis of the program.

During parsing, it is also suggested to construct a representation of the parsed program and store it in memory. In this way, it is possible to do the semantic analysis without parsing the same code again.

Abstract Syntax Tree (AST) is a tree structure used for the representation of the program in memory. The AST represents the abstract syntactic structure of the program. It is possible to make additional verification at the semantic level in the AST itself [Bet16].

Typically, a DSL must be integrated in an IDE. Since DSL is supported by the IDE features (e.g. syntax-aware editor, immediate feedback, incremental syntax checking, suggested corrections and auto-completion) it will be easier to learn, use, and maintain [Bet16].

2.3.1 XText

Xtext is an Eclipse tool that allows the implementation of programming languages, including DSLs. With the use of this tool the implementation becomes faster and allows to cover all aspects of the language infrastructure, namely the parser, the code generator and the interpreter.

With the use of Xtext, it is possible to begin the implementation of DSL without the creation of an AST. This creation and annotation of the rules for the AST construction are done automatically by Xtext. It will generate the lexer, the parser, the AST model and the AST construction [Bet16].

An Xtext-based grammar is specified through a notation that contains a set of parsing rules. These rules specify the concrete syntax of the language and also allow mapping with the abstract syntax. Each rule configures the structure of the domain language through a set of tokens. These tokens can be ID rule, string, value, integer value, reference value to other defined entity, special characters, etc. When a new Xtext project is created, several files corresponding to different aspects of the language are generated (Figure 2.5). The generated files are:

- main project file - defines the grammer, the runtime and the generated model;
- ide - compresses the details of the user interface, independent of Eclipse, used mainly for the external integration of the tool;
- tests - defines JUnit tests dependent from the main folder;
- ui - encompasses the Eclipse editor and features;
- ui tests - defines JUnit tests dependent from the user interface folder;

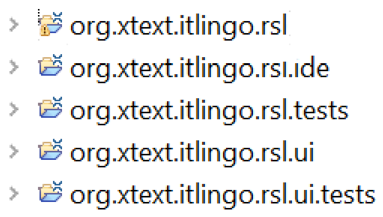

org.xtext.itlingo.rsl
org.xtext.itlingo.rsl.ide
org.xtext.itlingo.rsl.tests
org.xtext.itlingo.rsl.ui
org.xtext.itlingo.rsl.ui.tests

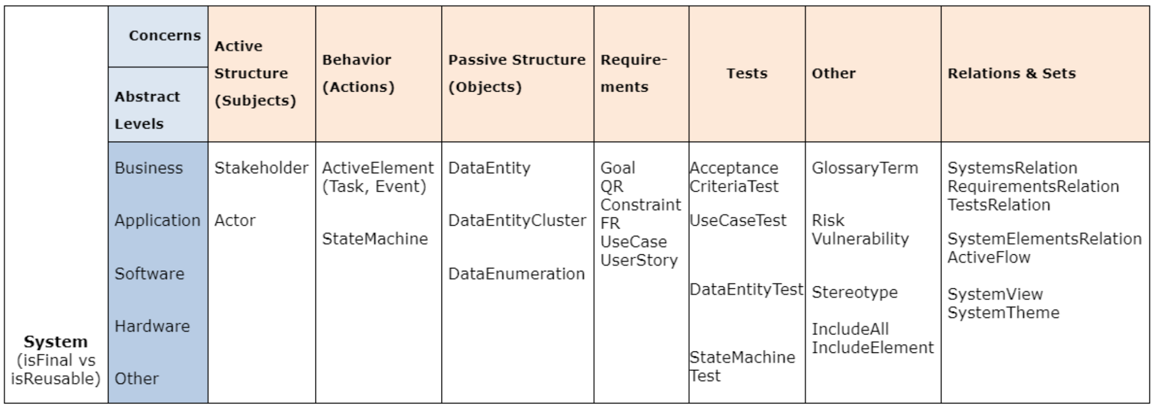
Figure 2.5: Xtext generated folders

When the grammar is already implemented, it is possible to configure the generation of artifacts by running Xtext MWE2 (Modeling Workflow Engine 2). During its execution are generated the artifacts related to the UI editor of the developed DSL and the AST is created. When completed it is possible to start an instance of Eclipse to create DSL specifications.

2.3.2 Requirement Specification Language

ITLingo research initiative intends to develop and apply rigorous specification languages for the IT domain, such as requirements engineering and testing engineering, with the RSL [SPS18]. RSL is a DSL that provides a comprehensive set of constructs that might be logically arranged into views according to specific concerns. These constructs are defined by *linguistic patterns* and represented textually according to concrete *linguistic styles*. RSL is a process- and tool-independent language, i.e., it can be used and adapted by different organizations with different processes or methodologies and supported by multiple types of software tools [Sil18]. However, in practice, RSL has been implemented with Xtext framework[2], so its specification is rigorous and can be automatically validated and transformed into multiple representations and formats. This paper focuses on the RSL constructs particularly supportive of use case approaches (e.g. actors, data entities and involved relationships).

RSL constructs are logically classified according to two perspectives (see Figure 2.6) [Sil18]: abstraction level and specific concerns they address. The abstraction levels are: business, application, software and hardware levels. On the other hand, the concerns are: active structure (subjects), behaviour (actions), passive structure (objects), requirements, tests, other concerns, relations and sets.



	Concerns / Abstract Levels	Active Structure (Subjects)	Behavior (Actions)	Passive Structure (Objects)	Require-ments	Tests	Other	Relations & Sets
	Business	Stakeholder	ActiveElement (Task, Event)	DataEntity	Goal QR Constraint FR UseCase UserStory	AcceptanceCriteriaTest	GlossaryTerm	SystemsRelation RequirementsRelation TestsRelation
	Application	Actor	StateMachine	DataEntityCluster		UseCaseTest	Risk Vulnerability	SystemElementsRelation ActiveFlow
	Software			DataEnumeration		DataEntityTest	Stereotype	SystemView SystemTheme
	Hardware						IncludeAll IncludeElement	
System (isFinal vs isReusable)	Other					StateMachine Test		

Figure 2.6: Classification of RSL constructs: abstraction levels versus RE specific concerns (extracted from [Sil18])

From a syntactical perspective, any construct can be used in any type of system regardless of its abstraction level. That means, for example, that it is possible to use a *DataEntity* construct at *Application* or *SoftwareSystem* levels but also at Business or even *HardwareSystem* levels. However, the use of a *DataEntity* at Business level shall be more general and incomplete (e.g., without data attributes specification) in comparison with its use at *Application* or *SoftwareSystem levels*, that shall be more detailed (e.g., including data attributes).

[2]https://www.eclipse.org/Xtext/

2.4 Test Case Generation Techniques

Test cases for systems with a complex domain are usually done manually and derived from functional requirements in natural language. An important factor is to ensure clear traceability between requirements and test cases. As a consequence, the definition of test cases ends up consuming a lot of time and being challenging. In this context, generating test cases from requirements in addition to reducing test costs helps to ensure that test cases cover all requirements.

Some approaches require that the system is captured in UML behavioural models, such as activity diagrams, statecharts, and sequence diagrams [CWI15]. For this reason, it is important to analyse existing methods for this purpose and able to evaluate and identify which is the most advantageous.

2.4.1 Test-duo

Hsieh et al. [CYHC13] present the Test-duo framework for generation and execution of acceptance tests from use case specifications. To specify use cases of the system under test (SUT) with the the framework test-duo, the authors recommend a set of steps, namely: [CYHC13]:

- The tester shall create explicit use cases when noting each step of the test case;
- The tester shall prepare a set of input data;
- The tester shall write the expected results generator;
- The framework iteratively generates single step tests and delegates them to the acceptance testing platform;
- The framework records each sequence of steps that ends in a state where a single step test fails and returns them as a set of unique test cases;

Although the first steps are performed manually by a tester, other steps are supported by a plugin for Eclipse [ECL18] called Acceplipse. Acceplipse allows to annotate use cases and translate them into a set of Prolog facts for input data and use case steps as Prolog relations where the tester modifies them into an expected results generator. These relations are queried by the framework when executing the last two steps automatically.

The Test-duo framework is organized into the test director and the test driver component, as shown in Figure 2.7. When consulting the Prolog relations generated through the annotation, the test director, a Prolog program, is responsible for generate test steps and all input data and expected results and directs them to the test driver. The test driver is implemented on top of the Robot[3] framework as a fixed script that receives commands and involves the SUT to execute keywords and send test results back to the test director.

The annotation mentioned above begins with the annotation of use cases as a structured program, following the identification of keywords and variables and the association between them.

[3]http://robotframework.org/

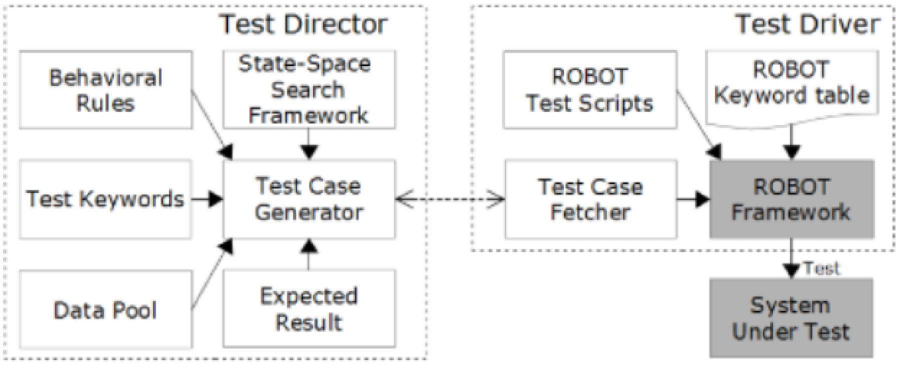


Figure 2.7: Test-duo components (extracted from [CYHC13])

2.4.2 TestMEReq

Moketar et al. [MKS+16] introduce a tool for early validation of requirements, TestMEReq. The purpose of this tool is to support the requirements engineer through the generation of abstract test components, namely test cases and mock-up prototypes of the user interface.

Following the process presented in Figure 2.8, the test cases are generated from the semi-formal abstract model named Essential Use Case (EUC). An EUC is a structured narrative expressed in an application and user domain language. The process begins with capturing the requirements as a user story or a use case scenario. Then the requirements are transformed into an EUC model through TestMEReq, and the requirements are analyzed by an EUC pattern library to generate the model. In the next step, a set of test cases is generated through the EUC model with the aid of another library of requirements testing patterns, with a specific syntax, in order to ensure consistency and uniformity. Then a set of test cases is derived from requirements testing using a library of test case patterns.

The above-mentioned libraries for the abstract tests generation arose from the collection and categorization of sentences from various types of requirements language and storing them as essential interactions. The patterns library for test cases consists of some key components such as the test case ID, test requirements, description, pre-conditions, input data, steps, and expected results.

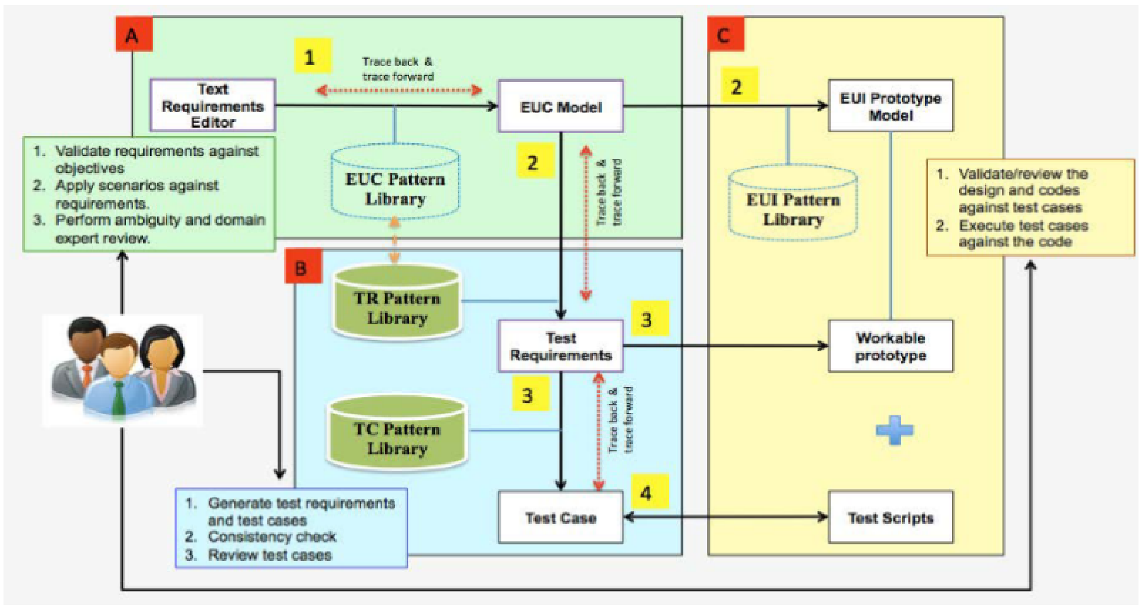

A
1
Trace back & trace forward
Text Requirements Editor
EUC Model
EUC Pattern Library
2
Trace back & trace forward
1. Validate requirements against objectives
2. Apply scenarios against requirements.
3. Perform ambiguity and domain expert review.
B
TR Pattern Library
Test Requirements
3
TC Pattern Library
3
Trace back & trace forward
4
Test Case
1. Generate test requirements and test cases
2. Consistency check
3. Review test cases
C
2
EUI Prototype Model
EUI Pattern Library
1. Validate/review the design and codes against test cases
2. Execute test cases against the code
Workable prototype
Test Scripts

Figure 2.8: TestMEReq process (extracted from [MKS+16])

2.4.3 UMTG

Wang et al. [CWI15] propose the UMTG (Use Case Modeling for System Tests Generation), an approach that automatically generates executable test cases from use cases and the domain model.

UMTG uses Natural Language Processing (NLP) techniques, a restricted form of use case specification, to extract behavioral information that allows the automation of tests, named Restricted Use Case Modeling (RCUM).

Initially, requirements are elicited with the RCUM, and the domain model is created manually as a UML class diagram. UMTG automatically checks if the model includes all the entities mentioned in the use cases. When the model is complete, the textual description of the pre-conditions and post-conditions is extracted automatically. Next, UMTG processes use cases with the Object Constraint Language (OCL) to generate use case test models for each use case.

The software engineer provides a mapping table that maps the descriptions of the high-level operations and the test inputs that must be performed by the test cases. The executable test cases are then generated from the mapping table.

2.4.4 Use Case Maps

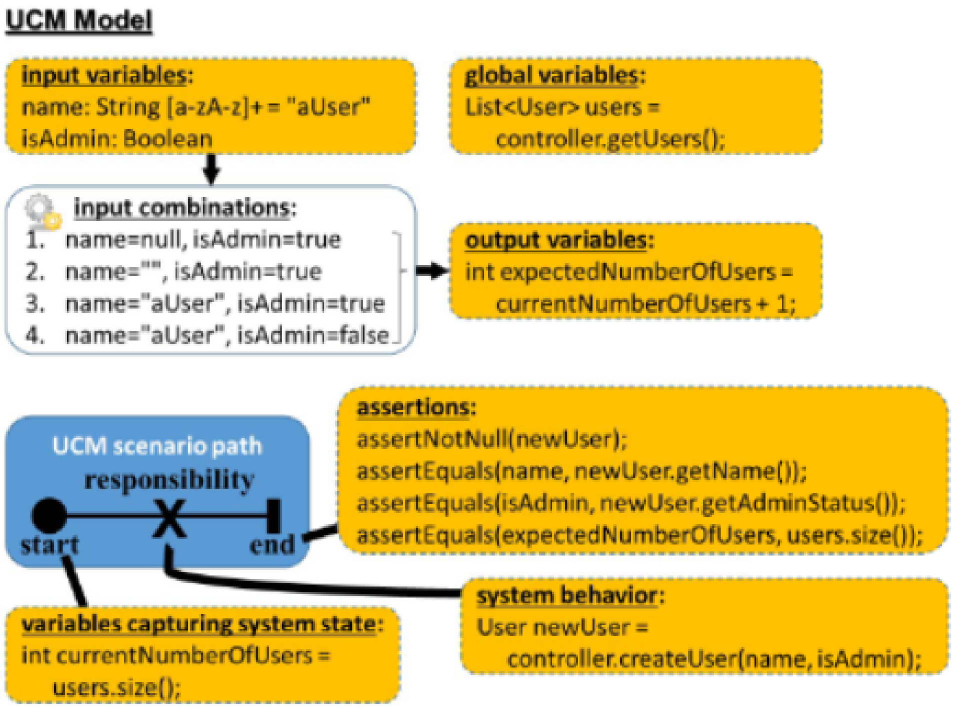

UCM Model
input variables:
name: String [a-zA-z]+ = "aUser"
isAdmin: Boolean
global variables:
List<User> users =
controller.getUsers();
input combinations:
1. name=null, isAdmin=true
2. name="", isAdmin=true
3. name="aUser", isAdmin=true
4. name="aUser", isAdmin=false
output variables:
int expectedNumberOfUsers =
currentNumberOfUsers + 1;
UCM scenario path
responsibility
start
end
assertions:
assertNotNull(newUser);
assertEquals(name, newUser.getName());
assertEquals(isAdmin, newUser.getAdminStatus());
assertEquals(expectedNumberOfUsers, users.size());
system behavior:
User newUser =
controller.createUser(name, isAdmin);
variables capturing system state:
int currentNumberOfUsers =
users.size();

Figure 2.9: UCM model example (extracted from [BM17])

Boucher et al. [BM17] investigated the transformation of workflow models with the Use Case Maps (UCM) notation, such as the example shown in figure 2.9, in end-to-end acceptance test cases that can be automated through the JUnit[4] test framework. Workflow models expressed in UCM provide a high-level description of an application's feature by focusing on causal relations between workflow steps and combining workflows into a high-level system view where behavior overlaps structural elements. A transversal mechanism analyzes the UCM model based on scenario definitions with the pre-conditions and post-conditions expected, making the definitions in a test suite regressive at the level of the UCM.

[4]https://junit.org/junit5/

2.5 Testing Automation Tools

Similar to some of the tools and approaches described in Section 2.4 , this research needs a tool that can automate the execution of tests.

Test cases can be executed manually by the tester or automatically by a test automation tool. When the test case is executed manually, the tester must perform all test cases, having to repeat the same tests several times throughout the product life cycle. On the other hand, when the test cases are run automatically, there is the initial effort to develop test scripts, but from there, the execution process will be automatic. So, if a test case has to run many times, the automation effort will be less than the effort of frequent manual execution. In this section, four tools were selected to be analyzed, tested and compared, with the intention of verifying which would be the most appropriate for this research.

The tests performed on these tools were applied on FEUP SIGARRA[5], the FEUP information and process management service.

2.5.1 Gauge

Gauge [Gau18] is a test automation framework, developed by ThoughtWorks, also creator of Selenium[6]. In addition to being cross-platform and open-source, Gauge supports a large variety of languages including Java, C#, Python and Javascript, which allows it to be used in any language and IDE. The tests created for this framework are written in Markdown, a lightweight markup language with simple text formatting syntax.

In the Listing 2.2, the specification of the test case based on the MarkDown syntax is presented. This specification is initiated by Specification Heading that can be tagged using Tags. Each specification must have one or more scenarios that represent a single flow that can also be associated with tags. In each scenario, there are one or more steps that represent the executable components of the specification and each step has underlying implementation code. The values written between quote marks are parameters that are passed to the implementation of the step as language specific structure.

[5]https://sigarra.up.pt/feup/pt/web_page.inicial
[6]https://www.seleniumhq.org/

```
Sigarra Login Box Specification
================================
tags: SIGARRA, browser

SIGARRA Login box should be displayed
======================================
tags:login

* Navigate to "https://sigarra.up.pt/feup/pt/web_page.Inicial"
* Find "caixa-validacao-conteudo" box and see if it is displayed
```

Listing 2.2: Gauge specification example

2.5.2 FitNesse

FitNesse [Fit18] is a lightweight, open-source framework that makes it easy for software teams to collaboratively define Acceptance Tests, web pages containing simple tables of inputs and expected outputs and to run those tests and see the results. FitNesse is an automated test tool, wiki, and web server all rolled into one application [HK06]. For instance, table 2.1 shows a simple table used to check if it is possible to login in SIGARRA portal.

Table 2.1: FitNesse specification example

url	**checkLoginBox**
https://sigarra.up.pt/feup/pt/web_page.Inicial	caixa-validacao-conteudo

2.5.3 Cucumber

Cucumber is a BDD automated acceptance test tool that allows users to write the specification of application features and user scenarios in an easily readable and understandable format for business analysts, developers, testes and others [SPS18]. It works with Ruby, Java, .NET, Flex or web application written in any languages. Cucumber allows users to describe the specification or features in a plain text in the form of Given-When-Then format, while the executable part of the test cases is written in Java code. Code snippet 2.3 shows an example of a Cucumber acceptance test or feature in Gherkin.

```
Feature: Access Sigarra

Scenario: Login functionality exists
  Given I have open the browser
  When I open Sigarra website
  Then Login button should exists
```

Listing 2.3: Cucumber specification example

Gherkin provides a lightweight framework to document examples of behaviours that stakeholders want in a format understandable by both stakeholders and Cucumber itself. Gherkin files use the feature extension and are saved as plain text with the support of a set of special keywords. Each feature is a collection of scenarios defined by a sequence of steps and following the Given-When-Then (GTW) rule. Given is used to set the context where the scenario happens, When defines when it will interact with the system, and Then checks if the outcome of that interaction was what was expected [WHT17].

2.5.4 ROBOT Framework

Robot Framework [Rob18] is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). It has a easy-to-use keyword-driven testing approach. Keywords are divided into higher-level user keywords and lower-level library keywords. The available keywords of Robot Framework are defined in libraries. This framework does not require any kind of implementation, since it is possible to use keywords with implicit implementations (with the use of those specific libraries such as Selenium[7]). The standard libraries are distributed with Robot Framework and the external libraries are released in separate packages [TPK11].

In the code snippet presented in 2.4 is used the keyword library that allows to manipulate web pages through the tool Selenium. When launched, Robot Framework will execute the test cases in order given in the test script and generate log files for results in HTML format.

```

*** Settings ***
Documentation   SIGARRA Acceptance Test
Library       Selenium2Library

*** Variables ***
${Login_Box}  caixa-validacao-conteudo

*** Test Cases ***
Login
 Open the browser on <https://sigarra.up.pt/feup/pt/web_page.Inicial>
 Page should contain element  ${LoginBox}
 ...

*** Keywords ***
Open the browser on <$(url)>
    Open Browser $(url)
```

Listing 2.4: Robot Framework specification example

[7]https://www.seleniumhq.org/

The script structure is simple and can be divided into four sections. The first section, *Settings*, where paths to auxiliary files and libraries used are configured. The second section, *Variables*, specifies the list of variables that are used, as well as the associated values. The third and most important section is the *Test Cases*, where test cases are defined. Lastly, the *Keywords* section define custom keywords to implement the test cases described in the Test Cases section. Among the sections mentioned above, only the *Test Cases* section is mandatory.

2.6 Discussion

As a result of the requirements development process it is produced the SRS, an agreement among stakeholders that describes the knowledge of the system under development. It contains multiple technical concerns of the system, which may include business requirements and user requirements (e.g. User Stories or Use Cases). SRS is the main information source for product's functional and nonfunctional requirements, which can be used throughout all the project life-cycle, facilitating the communication and the project management during the whole development process.

The section 2.2 presents the technique that will be approached throughout the research, namely model-based testing with the support of GUI testing tools. For this, software testing was briefly contextualized as well as the test levels that exist, giving greater focus to acceptance tests, since it is the test level that is considered through the developed solution.

By analyzing the formal language RSL, the advantages of this approach are visible, allowing the validation of the requirements and reduce the main problems related to the requirements.

A comparison between the related work is shown in Table 2.2 where it is concluded that the Test-duo tool and the workflow-based approach allow the generation of test cases and their execution, but do not promote the quality of the requirements. In addition, the Test-duo tool has a plug-in, developed by the same authors, that allows to support and simplify the manual process necessary for the use of the tool, as well as the integration of an automation tool, namely the Robot framework.

On the other hand, although it is not possible to run tests, the other approaches promote the quality of requirements through the type of specifications used that reduce ambiguity and inconsistency. TestMEReq also has several libraries developed by them to allow the generation of test cases.

With this in mind, the Test-duo and TestMEReq tools stand out, as they require less manual effort than the rest to achieve similar goals.

Table 2.2: Comparison of related work

	Test-duo	**TestMEReq**	**UMTG**	**Workflow Models**
Input data	Use Cases w/ annotations Pool of inputs Expected test results	Requirements as user stories	Requirements w/ RCUM Domain model Mapping table	Requirements defined w/ UCM model
Promote Requirements Quality	No	Yes	Yes	No
Test case generation	Yes	Yes	Yes	Yes
Tests Execution	Yes	No	No	Yes
Support Tools	ROBOT framework Acceplipse	Pattern Libraries		JUnit

In contrast to the tools and approaches investigated in Table 2.2, the tool that will be described in this research addresses all the comparative points, namely the promotion of the quality of requirements that is ensured by the use of the specification in the RSL language and the alignment with the test specification, the generation of test cases and the execution of the generated tests.

Table 2.3 presents the comparison between the testing automation tools analysed, taking into account their learnability, popularity, compatibility with the most popular programming languages, type of syntax and the need to implement the steps of the test cases .

Table 2.3: Comparison of Acceptance testing tools

	Gauge	**FitNesse**	**Cucumber**	**Robot**
Learnability	Easy	Complex	Easy	Medium
Popuparity	Small	Good	Good	Good
Specification Language	MarkDown	Tabular	Gherkin	Keywords
Implementation Languages	C#, Java, Javascript, Python, Ruby	Java, C++, C#, Python, Ruby, Delphi	Java, PHP, C#	Python, Java, C#
Step Implementation Integrated	No	No	No	Yes

After being tested, the conclusions drawn about Gauge were that, in fact, it is a simple and easy-to-learn tool, but once it is a relatively recent tool, it has a small community dimension and, consequently, lack of content usually provided by the community to solve some common problems.

Regarding the FitNesse tool, it is a very popular tool for the automation of acceptance tests and allows an easy documentation since it is a wiki server, however, the use of this wiki is mandatory. Although, within the range of tools selected, being the most commonly used tool by the community, it was concluded in this study that the language is very irregular for the purpose, since there is a great diversity of tables, each one with its own syntax. Moreover, when there are errors it is not always easy to understand what is wrong.

Cucumber uses Gherkin as a language for the specification of test cases, which is a great advantage, since it is written in plain text and in English and, therefore, understandable by everyone. In addition, it presents a large and active community which makes it easier to find solutions to the problems encountered. On the other hand, comparing with other tools, Cucumber needs a lot of code to make the specification executable.

Lastly, Robot Frarmework stands out for its powerful keyword-based language that does not require any kind of implementation and includes out-of-the-box libraries. However, the use of variables, imports, libraries, and even the keywords themselves can make the use of the tool a little more difficult for non-technical people.

From the direct comparison between the mentioned tools, the Robot tool is highlighted once it does not present any negative factor in comparison with the others. In addition, it is the only tool that does not require the implementation of the test code, which facilitates the automation process.

Chapter 3

RSL/Tests Language

The requirements can be represented in different ways, but the natural language is the most common and preferred [Dav05], being used in both documents and models. The problem is that the manual effort required to produce document requirements specifications is high and, given the large amount of information to be considered, this activity is error prone and time consuming.

To solve this situation, it would be advantageous to automate some of the manual tasks for Domain Analysis purposes, as well as, the verification of the extracted domain knowledge and the automatic generation of alternative requirements representations.

Given the purpose for which requirements specifications are created, it has been argued in [FS12] that simplified Natural Language Process (NLP) techniques can be used to support requirements specification by avoiding inconsistencies that are often difficult to detect. In addition, information extracted from natural language representations can be used to improve the quality of requirements, applying best practices and reducing ambiguity and inconsistency by crossing information extracted from lexical sources.

Initially, through the Requirements Specification Language (RSL) it was proposed an approach to the use of simplified natural language processing techniques to gather relevant information from natural language requirements specifications in ad-hoc and then extract knowledge from the encoded domain.

Later, it was designed a broader and consistent language, called "RSLingo RSL" (or just "RSL" for brevity), based on the design of the former languages, i.e., ProjectIT-RSL ([CVS06]), RSL-IL ([FS13a]), RSL-PL [FS13b], XIS* ([SSF$^+$07, RS14]), but also others such as i* ([Yu97]), Pohl ([Poh10]), and SilabREQ ([SSVM15]).

Recently, this language was extended in order to specify test cases directly from a RSL model. This test specification extension allows the construction of three different requirements test patterns, from the perspective of functional system tests, namely Domain Analysis, Use Case Testing and State Machine Testing [SPS18].

The approach described in chapter 4.3 was mainly based on the Use Case Testing test pattern, which allows the derivation of test scenarios from different workflows expressed in RSL Use Cases [SPS18]. However, besides the use of RSL there was also the need to extend it to meet the needs.

This chapter presents the main RSL constructs particularly extended in the scope of this research, most directly related with RSL/Tests extension, and in particular with *UseCaseTest* (as shown in Figure 3.2).

3.1 Requirements Specification

Figure 3.1 shows a partial view of the RSL metamodel that defines a hierarchy of requirement types, namely: goals, functional requirement, constraint, use case, user story and quality requirement.

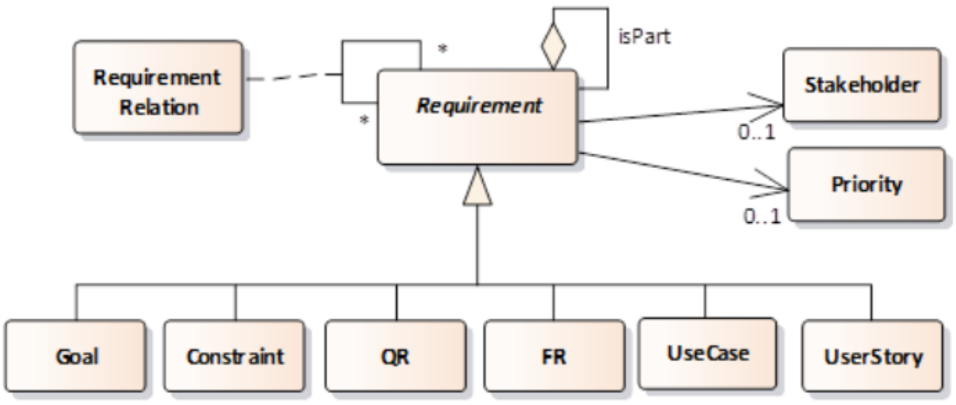


Figure 3.1: RSL partial metamodel: The hierarchy of requirements (extracted from [Sil18])

RSL specifications based on *Use Cases* can involve the definition of some views with the respective constructs and inherent relations:

- *DataEntity view*: defines the structural entities that exist in an information system, commonly associated to data concepts captured and identified from the domain analysis. A *Data Entity* denotes an individual structural entity that might include the specification of attributes, foreign keys and other checkdata constraints;
- *DataEntityCluster view*: denotes a cluster of several structural entities that present a logical arrangements among themselves;
- *Actor view*: defines the participants of *Use Cases* or *user stories*. Represent end-users and external systems that interact directly with the system under study, and in some particular situations can represent timers that trigger the start of some *Use Cases*;
- *Use Case view*: defines the *use cases* of a system under study. Traditionally a use case means a sequence of actions that one or more actors perform in a system to obtain a particular result [Jea15];

3.2 Tests Specification

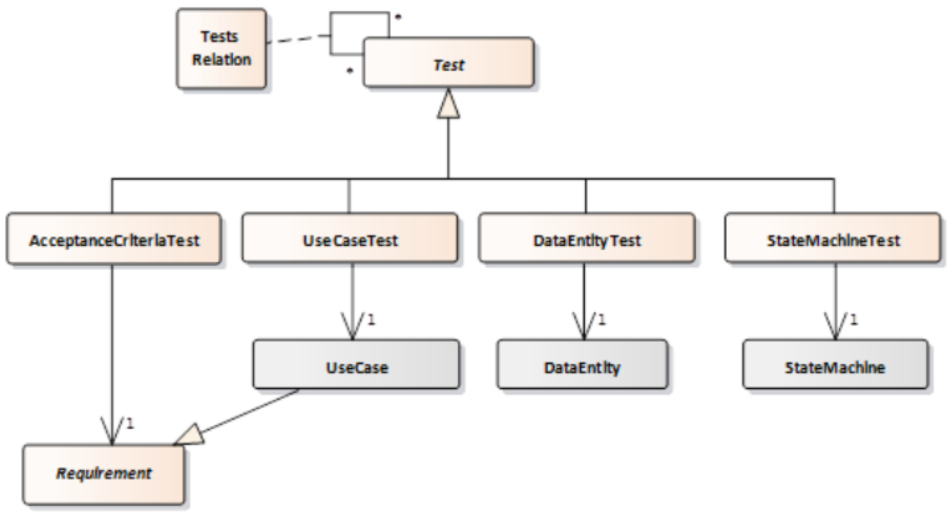


Figure 3.2: RSL partial metamodel: the hierarchy of Tests (extracted from [Sil18])

RSL supports the specification and generation of software tests, directly from requirements specifications. As showed in Figure 3.2, RSL provides a hierarchy of Test constructs and supports the specification of the following specializations of test cases [Sil18]:

- *DataEntityTest*: apply equivalence class partitioning and boundary value analysis techniques over the domains defined for the DataEntities [BQ15] in RSL *DataEntities*;
- *UseCaseTest*: explores multiple sequences of steps defined in RSL use cases' scenarios, and associates data values to the involved data entities;
- *StateMachineTest*: apply different algorithms to traverse the RSL state machines so that it is possible to define different test cases that correspond to valid or invalid paths through the respective state machine;
- *AcceptanceCriteriaTest*: define acceptance criteria based on two distinct approaches: scenario based (i.e., Given-When-Then pattern) or rule based; this test case is applied generically to any type of RSL Requirement

Regardless of these specializations, a *Test* shall be defined as *Valid* or *Invalid* depending on the intended situation. In addition, it may establish relationships with other test cases through the *TestsRelation*; these relationships can be further classified as *Requires*, *Supports*, *Obstructs*, *Conflicts*, *Identical*, and *Relates*.

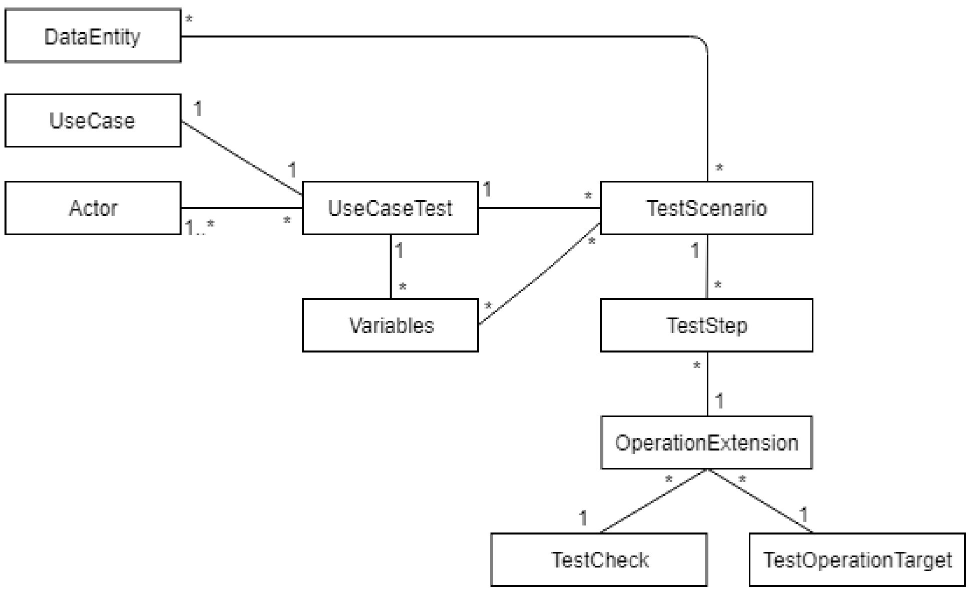


Figure 3.3: RSL/Tests extension metamodel

Regarding the different RSL test specification described, the *UseCaseTests* are the ones that best fit the acceptance testing. Figure 3.3 shows the structure and relations of *UseCaseTests*.

An *UseCaseTest* (Listing 3.1) inherits the data of the *UseCase* associated to it, including the *Actors*. Optionally, it is possible to add variables for testing purpose too.

```
UseCaseTest:
  'UseCaseTest' name=ID (nameAlias=STRING)? ':' type=TestType ('['
    'useCase' useCase=[UseCase | QualifiedName]
    ('actorInitiates' actorInitiates=[Actor | QualifiedName] )
    ('actorParticipates' actorParticipates+=RefActor)?
    ('background' background=[UseCaseTest | QualifiedName] )?
    (variables+=TestVariable)*
    (scenarios+=TestScenario)*
    (tags+=Tag)*
    ('description' description=STRING)?
  ']')?;
```

Listing 3.1: UseCaseTest RSL grammar

An *UseCaseTest* can have different *TestScenarios* (Listing 3.2). Each scenario must have at least one *TestStep* and, if applicable, the assignment of values to *DataEntities* and variables. Since *DataEntities* are entities integrated in the AUT, it may be useful to create instances of these entities and assign values to later be used in order to test cases related to those *DataEntities*. On the other hand, variables are temporary data that need to be passed between *TestSteps*, e.g. when it's needed some dynamic text presented in the GUI to then validate it, it must be saved that text in a variable.

```
TestScenario:
 'testScenario' name=ID (nameAlias=STRING)? ':' type=ScenarioType ('['
 ((isConcrete ?= 'isConcrete') | (isAbstract ?= 'isAbstract'))?
 ('variable'variable= [TestVariable | QualifiedName] ('withValues' '('
    variableTable= DataVariableValues ')'))?
  ('dataEntity' entity= [DataEntity | QualifiedName] ('withValues' '(' entityTable=
      DataAttributeValues ')'))?
  ('executionMode' mode=('Sequential'|'Parallel'))?
  ('description' description=STRING)?
  testSteps+= TestStep+
 ']')?;
```

Listing 3.2: TestScenario RSL grammar

The *TestStep* (Listing 3.3) is classified with an *StepOperationType* and eventually an *StepOperationSubType*. This operation types describe the action that will be performed in the respective step.

```
TestStep:
 'step' name=ID ':' type=StepOperationType (':' extension=OperationExtension)? ('['
  (simpleTestStep= SimpleTestStep );

OperationExtension:
  (subType=StepOperationSubType)
  ((target=TestOperationTarget)|(check=TestCheck))?;

enum StepOperationType: Actor_PrepareData | Actor_CallSystem | System_Execute |
    System_ReturnResult | Other | None;
enum StepOperationSubType: OpenBrowser | CloseBrowser | Reload | GetData | PostData
      | Select | Click | Over | Check | Other;
```

Listing 3.3: TestStep RSL grammar

There are four types of operations performed in *TestSteps*:

- **Actor_PrepareData**: it is expected that any type of data will be entered by the actor, such as text, passwords or even choose a file to upload;
- **Actor_CallSystem**: associates the actions performed by the actor in the application, e.g., click a button, select checkbox;
- **System_ReturnResult**: is used when it is necessary to collect application data to be stored in temporary variables that will normally be used for some type of verification;
- **System_Execute**: associates the actions that are executed by the system, e.g., open the browser and validations;

The *StepOperationSubTypes* are an extension for the previous types that specifies the action. These sub types are:

- **Open/CloseBrowser**: the action performed will open/close the browser;
- **Reload**: the action will reload the browser page;
- **GetData**: a specific data will be collected from the AUT;
- **PostData**: a specific data will be posted to the AUT;
- **Select/Click/Over**: specifies which action will be performed in an AUT element;
- **Check**: the action will verify some AUT content or response;

Finally, each step operation must have or a target (*TestOperationTarget*) or a verification (*TestCheck*) depending on the action associated (Listing 3.4).

```
TestOperationTarget:
  (type=OperationTargetType)
  ((variable+=[DataAttribute | QualifiedName] (','variable+=[DataAttribute |
      QualifiedName] )*)|
  ('(' content+=(STRING) (','content+=STRING)* ')'))?;
enum OperationTargetType : button | element | checkbox | listByValue | readFrom |
    writeTo;

TestCheck:
   (type=CheckType) ('('
   (variable=[DataAttribute | QualifiedName] '=' expected=[DataAttribute |
       QualifiedName])?
   ('text' (textVariable=[DataAttribute | QualifiedName]| textString=STRING))?
   ('timeout' (timeoutVariable=[DataAttribute | QualifiedName]| timeoutINT=
       DoubleOrInt) metric=Metric?)?
   ('limit' (limitVariable=[DataAttribute | QualifiedName]| limitINT=INT))?
   ('url' (urlVariable=[DataAttribute | QualifiedName]| urlString=STRING))?
   ('code' (codeVariable=[DataAttribute | QualifiedName]| codeString=STRING))?
   ')');
enum CheckType: textOnScreen | textOnElement | elementOnScreen | responseTime |
    variableValue | script | screen | Other | None;
```

Listing 3.4: TestOperation and TestCheck RSL grammar

If the action intends to interact with some GUI element, the *TestOperationTarget* will specify that element through the *OperationTargetType* which can be a button, a generic element, a check-box or a list. Additionally, the *OperationTargetType* is also used to specify if the element will be used to read or write. Besides that, the *TestOperationTarget* can have a description that is sent as a parameter through a variable value or a string.

The *TestCheck* defines the validation that will be performed in the step where was specified. There are seven types of validations (*CheckTypes*) where each of them has different parameters. Table 3.1 shows the set of validations available. Each *TestScenario* must end with a *TestStep* that has a *TestCheck*. When the test is executed, if the validation succeeds the test will pass.

Table 3.1: Test Step Validations

CheckType	**Parameter**	**Validation**
textOnScreen	text	checks if a specific text is presented in the GUI
textOnElement	text	checks if a specific text is presented in a specific element of the GUI;
elementOnScreen	limit?	checks if a specific element is presented in the GUI. If a limit is sent as parameter checks if a specific element appears less then the limit established.
responseTime	timeout	checks if the response time is less or equal than the given timeout;
variableValue	variable	checks if a variable value is equal to the expected value
	expected	
screen	URL	checks if the page represents the given URL
script	Code	uses a custom script to validate an unusual case

Chapter 4

Proposed Approach

Although it is considered a good practice to start testing activities early in the project, this is not frequently the common situation due to the traditional separation between the requirements and testing phases. This research intends to reduce this problem through a framework that encourages and supports both requirements and tests practices, namely by generating test cases from requirements or, at least, foster the alignment of such test cases with requirements.

The proposed approach (defined in Figure 4.1) begins with the (1) requirements specification that serves as a basis for the (2) test cases specification, which can be further (3) refined by the tester. Then, (4) tests scripts are generated automatically from the high-level test cases, and (5) associated the Graphical User Interface (GUI) elements. Finally, (6) these test scripts are executed generating a test report.

This set of phases covers the process of acceptance tests in interactive applications from the specification of requirements to the execution of tests. Applying the approach will establish an alignment between the specification of requirements and the specification of tests, in addition to increasing the processes automation.Besides the use and extension of the RSL grammar, the approach also uses support tools such as the Robot framework and Web Scrapper.

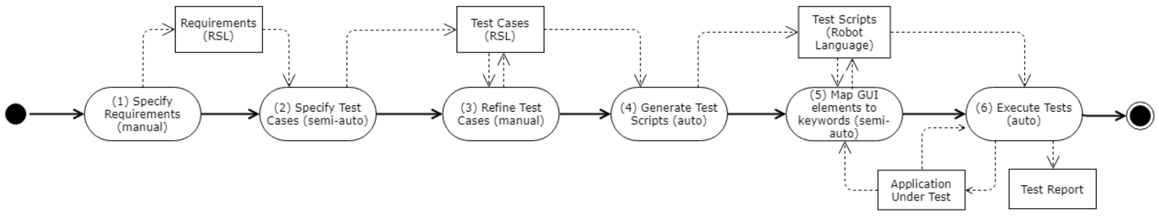

Requirements (RSL)
Test Cases (RSL)
Test Scripts (Robot Language)
(1) Specify Requirements (manual)
(2) Specify Test Cases (semi-auto)
(3) Refine Test Cases (manual)
(4) Generate Test Scripts (auto)
(5) Map GUI elements to keywords (semi-auto)
(6) Execute Tests (auto)
Application Under Test
Test Report

Figure 4.1: Proposed approach (UML activity diagram)

4.1 Specify Requirements

The first task of this approach is the requirements definition, an activity that usually involves the intervention of requirements engineers, stakeholders and eventually testers. After reaching a consensus, the specification of the requirements in RSL follows, through the constructs provided by the language that most fit the requirements domain. In this approach, the specification focuses on the most relevant RSL constructs at the application and software level, namely: *Actor, UseCase, DataEntity* and involved relationships. This task is usually performed by business analysts or by requirement engineers.

4.2 Specify Test Cases

Use Case Tests are derived from the various process flows expressed by a RSL *UseCase*. Each test contains multiple test scenarios. A test scenario encompasses of a group of test steps and shall be executed by an actor [SPS18].

From the requirement specifications, it is possible to specify test cases. *UseCaseTest* construct begins by defining the test set, including *ID*, *name* and the use case *type*. Then it encompasses the references keys [Use Case] indicating the *Use Case* in which the test is proceeding and [DataEntity] referring to a possible data entity that is managed [SPS18].

4.3 Refine Test Cases

The generated test cases may be subject to manual refinements (e.g., assign values to entities and create temporary variables), resulting in other test cases. The *DataEntities* and the temporary *Variables* are fundamental for data transmission between *TestSteps* involved in the test and are defined within *TestScenarios*.

Based on the RSL constructs it is possible to simplify the construction of the test cases reusing the information introduced in the requirements specification phase. It is in the scenarios that the *DataEntities* that will participate in the test are defined, as well as the temporary *Variables* that will be fundamental for data transmission between *TestSteps*.

The values of *DataEntities* and *Variables* are defined in a tabular format with several rows. In this way, when an attribute is associated with *N* values, it means that this scenario may be executed *N* times, once for each value in the table.

Each scenario is also formed by *TestSteps*. A *TestStep* can have four types (*Actor_PrepareData, Actor_CallSystem, System_Execute, System_ReturnResult*) and several *OperationExtensions* (see Table 4.1).

It is in the test cases that are introduced the fundamental concepts for the test scripts generation, which include test scenarios and test steps.

In the *UseCaseTest* specification the respective *UseCase* and *DataEntities* specifications are associated and temporary variables are initialized, such as the name of the product that will be searched and the number of expected results. Also in the *UseCaseTest*, the *TestScenarios* are specified where the values are assigned to the variables and inserted the *TestSteps*, which contain the necessary information for the test scripts.

4.4 Generate Test Scripts

Once the specification is complete, it follows the generation of the test scripts for the Robot tool. This generation process is based on relations established between the RSL specification and the syntax of the Robot framework. It is possible to make an association of the the RSL concepts with the Robot framework syntax and some of the keywords made available by the Selenium library (Table 4.1).

Table 4.1: Mapping between test case (TSL) and test scripts (Robot)

Step Type	**Operation Extension Type**	**Operation Extension**	**Keyword generated**
Actor_PrepareData	Input	readFrom	INPUT TEXT $locator $variable
Actor_CallSystem	Select	checkbox	SELECT CHECKBOX $locator
		list by value	SELECT FROM LIST BY VALUE $locator $value
	Click	button	CLICK BUTTON $locator
		element	CLICK ELEMENT $locator
	Over	-	MOUSE OVER $locator
System_ReturnResult	GetData	writeTo	$variable GET TEXT $locator
System_Execute	OpenBrowser	-	OPEN BROWSER $url
	CloseBrowser	-	CLOSE BROSER
	PostData	readFrom	INPUT TEXT $locator $variable
	Check	textOnPage	PAGE SHOULD CONTAIN $text
		elementOnPage	PAGE SHOULD CONTAIN ELEMENT $locator $msg? $limit?
		textOnElement	ELEMENT SHOULD CONTAIN $locator $text
		responseTime	WAIT UNTIL PAGE CONTAIN ELEMENT $locator $timeout?
		variableValue	$variable = $expected
		jScript	EXECUTE JAVASCRIPT $code

First, in *Actor_PrepareData* type, it is expected that any type of data will be entered by the actor, such as text, passwords or even choose a file to upload. The value of the data to be entered is acquired through the *DataEntities* defined previously in the *TestScenario* when the *OperationExtension* of the *TestStep* is 'readFrom' followed by the identifiers of the respective *DataEntity/Variable* attributes.

Second, the *Actor_CallSystem* type associates the actions performed by the actor in the application, e.g., click a button, select checkbox. In *OperationExtension* the type of action (e.g., Click, Select, Mouse Over) and the element on which such action takes place (e.g., button, checkbox, image) are identified.

Third, there is the *System_ReturnResult* that is used when it is necessary to collect application data to be stored in temporary variables that will normally be used for some type of verification. In this type of operation, the *OperationExtension* is 'writeTo' followed by the attribute of the variable where the value will be stored.

Finally, there is the *System_Execute* where the actions that are executed by the system, e.g., ‘OpenBrowser’ and ‘Check’, are associated. Each *TestScenario* must end with a Check in order to evaluate the success/insuccess of the test. The types of checks introduced are: text on element, element on page, text on element, response time, variable value or custom.

4.5 Map GUI Elements to Keywords

At this stage, there is the need to complete the test scripts generated in the previous phase with the locators, i.e. queries that return a single element which are used to locate the target GUI elements (e.g. GUI element identifier, xpath, CSS selector) [LCRT16]. Web applications interfaces are formed by sets of elements, namely, buttons, message boxes, forms, among other elements that allow to increase the User Interface (UI) interactivity. Each of these elements has a specific locator, which allows it to be recognized among all elements of the UI. During the acceptance testing activity, these elements are used to achieve a certain position defined by the test case. In order to automate the acceptance test cases generation and execution, it is necessary to identify these locators to be able to use the respective GUI elements during the execution of the test.

Since element identifiers usually do not follow any specific pattern, it becomes complex to do an automatic mapping. So, it is necessary the manual intervention of a technician to make the mapping between the GUI elements and the test scripts keywords. To establish this mapping the user can insert the corresponding identifiers of the UI elements in the test script or use a ‘point and click’ process (similar to the one in [PFTV05]) where he points the UI element on screen to capture the identifier of the clicked element. For this purpose, the Web Scrapper browser extension is used, which has a feature that allows to capture interactions with the application.

Following the workflow presented in Figure 4.2, the tester should select the element required for the test script. The tool returns the locator of the pointed elements. The tester after interacting with all the elements can add the returned locators manually into the test script previously generated. However, as alternative was also developed a pair of scripts that allow the mapping between locators and the test script. The *Mapping Script* creates a file with the map and the *Replacement Script* inserts the locators in the Robot test script. The tester after find all locators can extract through the Web Scrapper a JSON code that can be converted to a XML file by running the *Mapping Script*. This file will contain a map between the locators found with the missing locators in the test script generated. By running the *Replacement Script* the missing locators in the Robot test script will be replaced with the locators found with the Web Scrapper using the XML file generated with the *Mapping Script*.

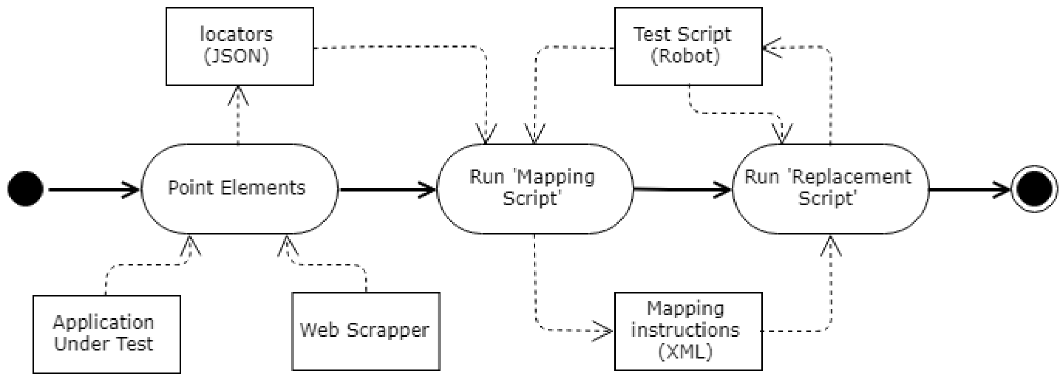


Figure 4.2: Map GUI Elements to keywords approach (UML activity diagram)

4.6 Execute Tests

The execution of the generated test script is performed using the Robot framework. At the command prompt the tester must run the following command: *robot [script_name].robot*. While executing, a browser instance will open performing automatically every steps specified. Meanwhile, the results of each test case will be displayed at the command prompt.

Chapter 5

Illustrative Example

In order to illustrate and discuss the suitability of the proposed approach, we applied it over an interactive web application. From a wide variety of possible web applications, it was selected the Web Store[1]. This application is a fake e-commerce site developed to practice test automation and covers the complete online shopping workflow. Figure 5.1 shows the screenshot of the home page of the store.

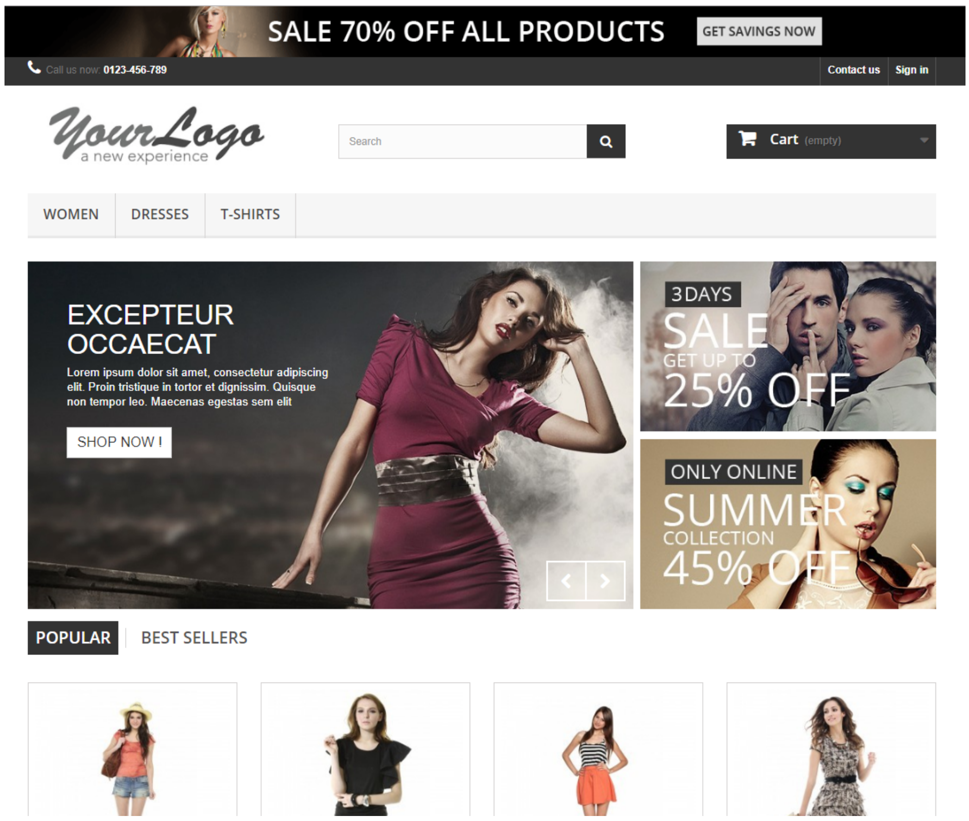

SALE 70% OFF ALL PRODUCTS
GET SAVINGS NOW
Call us now: 0123-456-789
Contact us
Sign in
YourLogo
a new experience
Search
Cart (empty)
WOMEN
DRESSES
T-SHIRTS
EXCEPTEUR OCCAECAT
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin tristique in tortor et dignissim. Quisque non tempor leo. Maecenas egestas sem elit
SHOP NOW !
3 DAYS SALE GET UP TO 25% OFF
ONLY ONLINE SUMMER COLLECTION 45% OFF
POPULAR
BEST SELLERS

Figure 5.1: Web Store application - Search Product

[1]http://automationpractice.com

To apply the developed research in this illustrative example, different entities were identified (Figure 5.2) and different RSL constructs were specified. For instance it was created the following *DataEntities, Actors and UseCases*:

DataEntity:

- e_Product (Listing 5.1)
- e_Cart
- e_Cart_Line
- e_Order
- e_Whishlist
- e_Wishlist_Line
- e_Review

DataCluster:

- c_Purchase
- c_Wishlist
- c_Review

Actor:

- a_Customer (Listing 5.1)
- a_Visitor

UseCase:

- uc_SignIn
- uc_AddToCart
- uc_Checkout
- uc_AddToWishlist
- uc_WriteReview
- uc_SearchProduct (Listing 5.1)

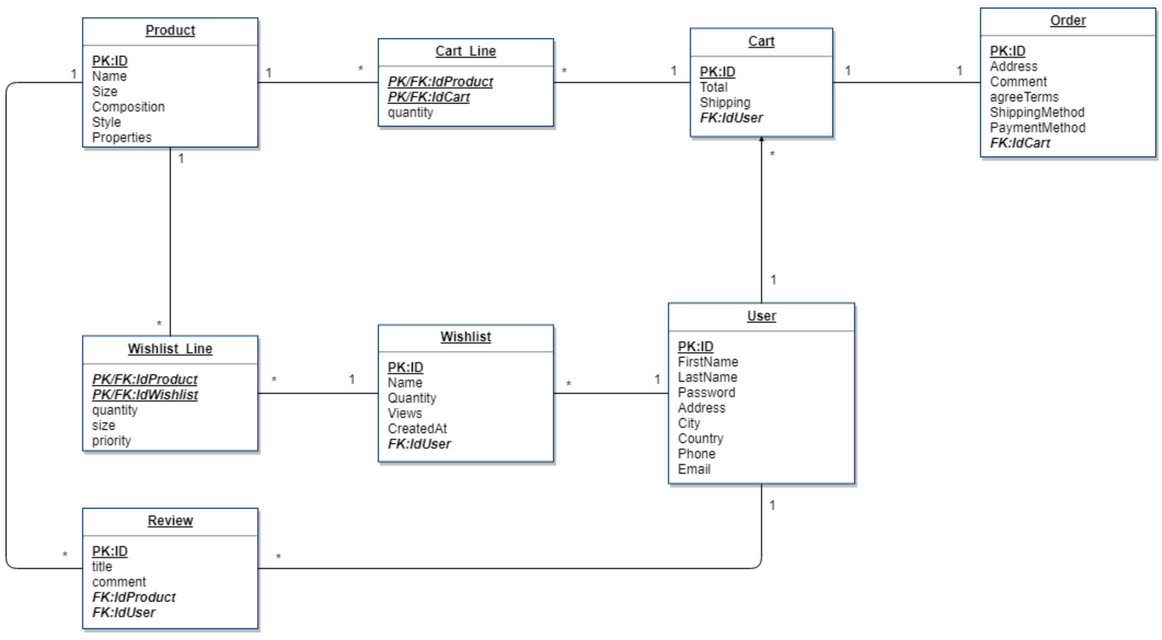


Figure 5.2: Web Store application - Entities

The 'Search Product' Use Case was selected to serve as a simple use case to support our discussion. In this Use Case, the user search for a product by name and the number of results must match the expected.

```
DataEntity e_Product "Product" : Master [
 attribute ID "ID" : Integer [isNotNull isUnique]
 attribute title "title" : Text [isNotNull]
 attribute price "Price" : Integer [isNotNull]
 attribute composition "Composition" : Text
 attribute style "Style" : Text
 attribute properties "Properties" : Text
 primaryKey (ID)]

Actor aU_Customer "Customer" : User [description "Customer uses the system"]

UseCase uc_Search "Search Products" : EntitiesSearch [
 actorInitiates aU_Customer
 dataEntity e_Product]
```

Listing 5.1: Example of a RSL specification of Data Entity Actor and UseCase

Regarding Listing 5.1, the requirements specification phase (section 4.1) is complete. Follows the specification of test cases (section 4.2) where is made a connection between the requirements and tests. A *UseCaseTest* is created based on the corresponding *UseCase*. After that, the test case must be refined (section 4.3) with *TestScenarios, TestSteps* and respective *DataEntities* and variables. For instance, Listing 5.2 shows a test case specified and refined with the necessary information to perform tests. In this case, two variables were associated. The first one (*'v1.search'*) is the keyword or phrase used to search products related with it while the second one (*'v1.expected'*) contains the number of results expected. In the *TestScenario* are represented the steps that are necessary to perform in on order to achieve the number of results and compare it with the expected number.

```
UseCaseTest t_uc_Search "Search Products" : Valid [
 useCase uc_Search actorInitiates aU_User
 description "As a User I want to search for a product by name or descripton"
 variable v1 [
  attribute search: String
  attribute expectedResults: String
 ]
 testScenario Search_Products :Main [
  isConcrete
  variable v1 withValues (
  | v1.search | v1.expectedResults +|
  | "Blouse"  | '1'    +|
  | "Summer"  | '3'     +|)
  step s1:Actor_CallSystem:Click element('Home')["The User clicks on the 'Home'
      element"]
  step s2:Actor_PrepareData:PostData readFrom v1.search ["The User writes a word or
       phrase in the search text field"]
  step s3:Actor_CallSystem:Click button('Search_Product')["The User clicks on the '
      Search' button"]
   step s4:System_Execute:Check elementOnScreen(limit v1.expectedResults)["The
       System checks if the number of results is the expected one"
 ]
]
```

Listing 5.2: Example of 'Search Products' test case RSL specification

After processing the available data, the code generator integrated in ITLingo-Studio will generate (section 4.4) the Robot test script resulting in a script similar to the one shown in Listing 5.3. However, there still missing the elements locators in the script so the Robot framework knows in which elements of the AUT must perform the command specified in the test script.

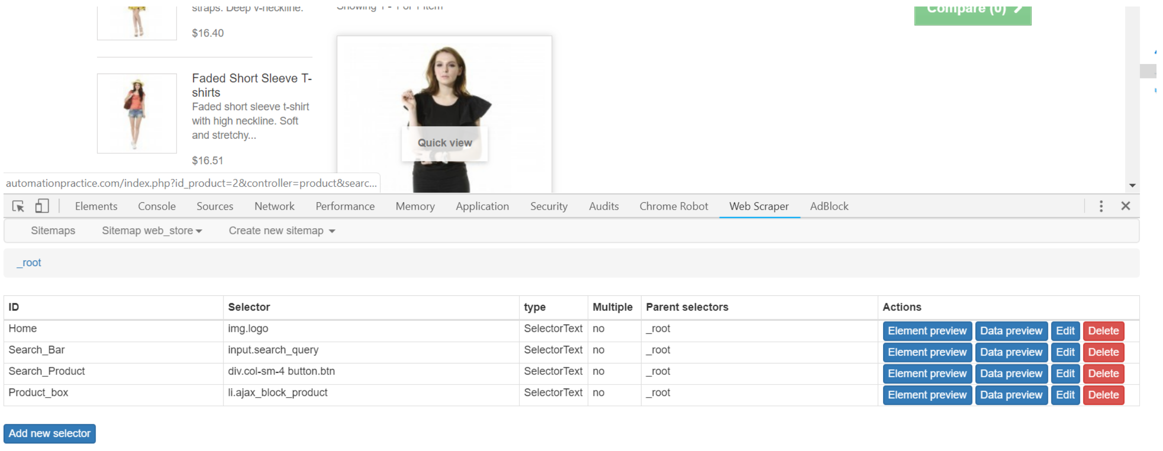
```
*** Variables ***
${search1}  Blouse
${search2}  Summer
${expectedResults1}  1
${expectedResults2}  4

*** Test Cases ***
Search_Products-Test_1
  [Documentation]   As a User I want to search for a product by name or descripton
  Click element [Home]
  Input text [Search_Bar] ${search1}
  Click button [Search_Product]
  Page Should Contain Element [Product_box] limit=${expectedResults1}


Search_Products-Test_2
  [Documentation]   As a User I want to search for a product by name or descripton
  Click element [Home]
  Input text [Search_Bar] ${search2}
  Click button [Search_Product]
  Page Should Contain Element [Product_box] limit=${expectedResults2}
```

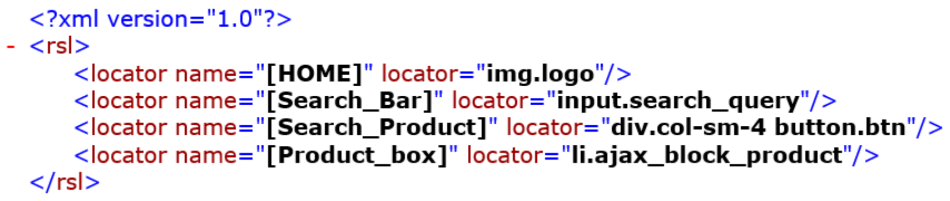
Listing 5.3: Generated Test Script example (in Robot)

To resolve this problem follows the next phase, the mapping of GUI elements in keywords (section 4.5). The tester access the AUT and with the help of the Web Scrapper he points to the desired elements. As shown in Figure 5.3, Web Scrapper save every locators (in this case a unique CSS selector) found until the date and allows to export in JSON code. After exporting to JSON, the tester can execute the *Mapping Script* that will create a XML file (Figure 5.4) which represents the mapping between the found locators and the missing locators in the Robot test script generated in the previous phase. For this, it's important that it be given the same description to the element in both Web Scrapper and test case specification. After that, it is possible to execute the *Replacement Script* that will complete the Robot test script with the locators through the replacement of the data provided by the XML file resulting in a script similar to Listing 5.4. For instance, "css:img.logo" is the CSS locator for the element that redirects the user to the "Home Page". Once complete, the test script will be able to be executed.



ID	Selector	type	Multiple	Parent selectors	Actions
Home	img.logo	SelectorText	no	_root	Element preview Data preview Edit Delete
Search_Bar	input.search_query	SelectorText	no	_root	Element preview Data preview Edit Delete
Search_Product	div.col-sm-4 button.btn	SelectorText	no	_root	Element preview Data preview Edit Delete
Product_box	li.ajax_block_product	SelectorText	no	_root	Element preview Data preview Edit Delete

Figure 5.3: Web Scrapper



```
<?xml version="1.0"?>
- <rsl>
      <locator name="[HOME]" locator="img.logo"/>
      <locator name="[Search_Bar]" locator="input.search_query"/>
      <locator name="[Search_Product]" locator="div.col-sm-4 button.btn"/>
      <locator name="[Product_box]" locator="li.ajax_block_product"/>
  </rsl>
```

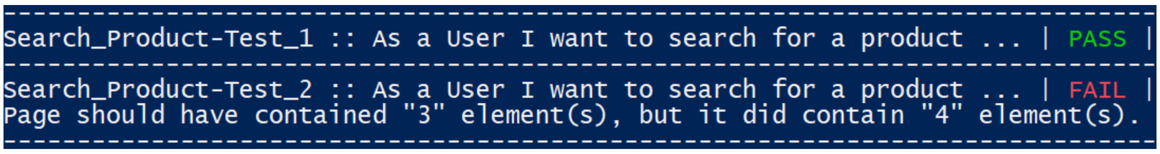
Figure 5.4: XML file - Map between locators and test script

```
*** Variables ***
${search1}  Blouse
${search2}  Summer
${expectedResults1}  1
${expectedResults2}  4

*** Test Cases ***
Search_Product-Test_1
  [Documentation]   As a User I want to search for a product by name or descripton
  Click Element css:img.logo
  Input text css:input.search_query ${search1}
  Click button css:div.col-sm-4 button.btn
  Page should contain element css:li.ajax_block_product limit=1

Search_Product-Test_2
  [Documentation]   As a User I want to search for a product by name or descripton
  Click Element css:img.logo
  Input text css:input.search_query ${search2}
  Click button css:div.col-sm-4 button.btn
  Page should contain element css:li.ajax_block_product limit=3
```

Listing 5.4: Test Script with GUI elements xpath (in Robot)



```
------------------------------------------------------------------------------
Search_Product-Test_1 :: As a User I want to search for a product ... | PASS |
------------------------------------------------------------------------------
Search_Product-Test_2 :: As a User I want to search for a product ... | FAIL |
Page should have contained "3" element(s), but it did contain "4" element(s).
------------------------------------------------------------------------------
```

Figure 5.5: Result of the test case execution

Once the script is completely filled in, the tests are executed (section 4.6) and the test results are displayed, as shown in Figure 5.5. Regarding the search for a product use case example, when searching for products associated to the word "Blouse", the test returned one result as expected and so, the test succeeded. On the other hand, when searching for products related to the word "Summer", the test returned 4 products which is different from the expected result (3 products) and so, the test failed.

Illustrative Example

Chapter 6

Conclusion

Model-based testing is a technique used by many approaches for bases common testing tasks, including test case generation through an abstract representation of the application under test. This technique have substancial appeal and it was already proved by different studies that testing a variety of applications has been met with success when MBT was employed [EFW02].

This research work describes a new approach that applies a MBT technique for generation and execution of acceptance test cases. This approach uses as a model of the AUT an intermediate requirements representations based on the formal language RSL By using a well-structured requirements specification based on RSL, it is granted better quality of requirements due to the fact of the RSL produce SRS in a systematic, consistent and rigorous way, removing problems such as ambiguity and incorrectness. On the other hand, by using requirements specifications it is established an alignment between the requirements and the tests that will be generated and it is implicitly initiated the testing process at the beginning of the project.

6.1 Final Discussion

Since the RSL language was not prepared to generate acceptance test cases, some constructs already in the grammar had to be adapted and modified and new concepts and rules introduced. In addition of the use and extension of RSL grammar, this approach used others support tools such as the Robot Framework and the WebScrapper.

Despite the fact that more automation has been brought to the testing practice, human intervention continues to be imperative. For acceptance tests execution automation it is mandatory to specify the localization of the GUI elements of the AUT, so the framework responsible to execute tests knows where to interact. To collect this data it is necessary manual intervention, even if it is just to point the elements. In order to solve this problem, it was introduced the browser extension WebScrapper turning the process more simple and interactive. However, even with the use of this extension it is necessary to add comments or description to each locator found to create a link with the test script, resulting in an unexpected manual effort.

Comparing with other approaches previously described in section 2.4 and presented in Table 6.1, this approach is the only capable of supporting the three main comparative points, namely if the approach promotes requirements quality, if supports test case generation and if can execute automatically. The RSL language itself promotes the requirements quality by reducing the ambiguity and the incompleteness of requirements. With the RSL requirements specification, it is possible to generate test cases, although they still need refinement. Lastly, the test execution is ensured by the Robot framework that can execute the test scripts generated by the approach.

Table 6.1: Comparison between related work and RSL/Tests Language

	Test-duo	**TestMERQ**	**UMTG**	**Workflows Models**	**RSL/Tests language**
Input Data	Use Cases w/annotations Pool of inputs Expected test results	Requirements as User Stories	Requirements w/ RCUM Domain Model Mapping table	Requirements defined w/ UCM model	RSL specifications
Promote Requirements Quality	No	Yes	Yes	No	Yes
Test Case Generation	Yes	Yes	Yes	Yes	Yes
Tests Execution	Yes	No	No	Yes	Yes
Support Tools	Robot framework Acceplipse	Pattern Libraries		JUnit	Robot Framework WebScrapper

With the purpose of testing the effectiveness of the approach developed, it was applied the entire process in an interactive application. From the requirements specification to test execution, the approach proved to be able to achieve the overall goals defined initially resulting in an end-to-end integration.

6.2 Future Work

This dissertation leaves an open door to improve even further the automation of the described approach.

First, it would be useful the creation of a browser extension similar to WebScrapper but dedicated to this cause. For instance, if the hypothetical extension allows the importation of XML files (containing the list of the desired elements locators) and the exportation of the resulting map can remove the undesired manual effort to align the elements locators gathered with the test script.

Second, automate further the overall process by automatically generate the test specifications from RSL specifications.

Finally, generate test scripts that may be executed through other test automation tools, such as Gherkin/Cucumber[1].

[1]https://cucumber.io/

References

[ACRPV05] J. C. P. Faria A. C. R. Paiva, N. Tillmann and R. F. A. M. Vidal. Modeling and Testing Hierarchical GUIs. *Proceedings of the 12th International Workshop on Abstract State Machines*, 2005.

[AO16] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[Bet16] L. Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016.

[BM17] M. Boucher and G. Mussbacher. Transforming Workflow Models into Automated End-to-End Acceptance Test Cases. *Proceedings - 2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering, MiSE 2017*, pages 68–74, 2017.

[BQ15] A. Bhat and S.M.K. Quadri. Equivalence class partitioning and boundary value analysis - A review. *2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2015.

[Chr08] J. Christie. The Magazine for Professional Test Automation - Do it make sense? 2008.

[Coc01] A. Cockburn. *Writing Effecive Use Cases*. Addison-Wesley, 2001.

[CVS06] D. Ferreira C. Videira and A. Rodrigues Da Silva. A linguistic patterns approach for requirements specification. *Proceeding 32nd Euromicro Conference on Software Engineering and Advanced Applications (Euromicro'2006), IEEE Computer Society*, 2006.

[CWI15] A. Goknil L. Briand C. Wang, F. Pastore and Z. Iqbal. Automatic Generation of System Test Cases from Use Case Specifications. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 385–396, 2015.

[CYHC13] C. H. Tsai C. Y. Hsieh and Y. C. Cheng. Test-Duo: A framework for generating and executing automated acceptance tests from use cases. *2013 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings*, pages 89–92, 2013.

[Dav05] A. Davis. *Just Enough Requirements Management: Where Software Development Meets Marketing*. Dorset House Publishing, First edition, 2005.

[ECL18] Open inovation community - eclipse ide | the eclipse foundation. Eclipse Foudnation, available in `https://www.eclipse.org/`, June 2018.

[EFW02] Ibrahim K. El-Far and James A. Whittaker. *Model-Based Software Testing*. American Cancer Society, 2002.

[Fit18] Front page. FitNesse, available in http://docs.fitnesse.org/FrontPage, June 2018.

[FS12] D. De Almeida Ferreira and A. 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.

[FS13a] D. De Almeida Ferreira and A. Rodrigues Da Silva. RSL-IL: An interlingua for formally documenting requirements. *2013 3rd International Workshop on Model-Driven Requirements Engineering, MoDRE 2013 - Proceedings*, (July):40–49, 2013.

[FS13b] D. De Almeida Ferreira and A. Rodrigues Da Silva. RSL-PL: A linguistic pattern language for documenting software requirements. *2013 3rd International Workshop on Requirements Patterns, RePa 2013 - Proceedings*, pages 17–24, 2013.

[Gau18] Gauge | thoughworks. ThoughWorks, available in https://www.gauge.org/, June 2018.

[GE17] V. Garousi and F. Elberzhager. Test Automation: Not Just for Test Execution. *IEEE Software*, 34(2):90–96, 2017.

[Gli14] M. Glinz. A Glossary of Requirements Engineering Terminology. (May):116, 2014.

[GP16] J. Esparteiro Garcia and A. C. R. Paiva. REQAnalytics: A Recommender System for Requirements Maintenance. *International Journal of Software Engineering and its Applications 10(129):140*, 2016.

[Gra02] D. Graham. Requirements and Testing : Myths, Seven Missing-link. *Engineering*, (October):15–17, 2002.

[HBL09] S. Hansen, N. Berente, and K. Lyytinen. *Requirements in the 21st Century: Current Practice Emerging Trends*, volume 14, pages 44–87. 01 2009.

[HCG16] S. Hotomski, E. B. Charrada, and M. Glinz. An Exploratory Study on Handling Requirements and Acceptance Test Documentation in Industry. *Proceedings - 2016 IEEE 24th International Requirements Engineering Conference, RE 2016*, pages 116–125, 2016.

[HH08] B. Haugset and G. K. Hanssen. Automated Acceptance Testing: A Literature Review and an Industrial Case Study. *Agile 2008 Conference*, pages 27–38, 2008.

[HK06] A. Holmes and M. Kellogg. Automating functional tests using selenium. *Proceedings - AGILE Conference, 2006*, 2006:270–275, 2006.

[IEE98] Ieee recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, Oct 1998.

[Int10] International Software Testing Qualification Board. Standard glossary of terms used in Software Testing. (July):20, 2010.

[IST] Istqb glossary. ISTQB, available in https://glossary.istqb.org/.

[IST15] *ISTQB® Foundation Level Certified Model-Based Tester Syllabus*. ISTQB, 2015.

[Jea15] I. Jacobson and et al. Object oreinted software engineering: A use case driven approach. *Addison-Wesley*, 2015.

[Kov98] B. Kovitz. Pratical software requirements: Manual of content and style. manning., 1998.

[LCRT16] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. *Approaches and Tools for Automated End-to-End Web Testing*, volume 101. Elsevier Inc., 1 edition, 2016.

[Lin05] C. Lindquist. Fixing the Software Requirements Mess. *CIO Magazine*, (November), 2005.

[LQ10] A. de Lucia and A. Qusef. Requirements engineering in agile software development. *Journal of Emerging Technologies in Web Intelligence*, 2(3):212–220, 2010.

[MB-18] Model-based testing. Microsoft, available in https://msdn.microsoft.com/en-us/library/ee620469.aspx, June 2018.

[MKS$^+$16] N.A. Moketar, M. Kamalrudin, S. Sidek, M. Robinson, and J. Grundy. TestMEReq: Generating abstract tests for requirements validation. *Proceedings - 3rd International Workshop on Software Engineering Research and Industrial Practice, SER and IP 2016*, pages 39–45, 2016.

[MP14] R. M. L. M. Moreira and A. C. R. Paiva. PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-based GUI Testing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 863–866, New York, NY, USA, 2014. ACM.

[MPNM17] R. M. L. M. Moreira, A. C. R. Paiva, M. Nabuco, and A. Memon. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Softw. Test., Verif. Reliab.*, 27(3), 2017.

[NT11] K. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2011.

[Pai07] A. C. R. Paiva. *Automated Specification-based Testing of Graphical User Interfaces*. PhD thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2007.

[Pat01] R. Patton. *Software Testing*. Sams, 2001.

[PFTV05] A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal. A model-to-implementation mapping tool for automated model-based GUI testing. In *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, pages 450–464, 2005.

[PFV07] A. C. R. Paiva, C. P. Faria, and R. F A M Vidal. Towards the Integration of Visual and Formal Models for GUI Testing. 190:99–111, 2007.

[Poh10] K. Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, First edition, 2010.

[PR15] K. Pohl and C. Rupp. *Requirements Engineering Fundamentals: A study guide for the Certified Professional for Requirements Engineering Exam Foundation Level/IREB compliant*. Rocky Nook, Second edition, 2015.

[PV17] A.C.R. Paiva and L. Vilela. Multidimensional test coverage analysis: Paradigm-cov tool. 20:633–649, 2017.

[Rob18] Robot framework. Robot Freamework Foudnation, available in http://robotframework.org/, June 2018.

[RR06] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, 2nd edition edition, 2006.

[RS14] A. Ribeiro and A. Rodrigues Da Silva. Evaluation of xis-mobil, a domain specific language for mobile application development. *Journal of Software Engineering and Applications, 7(11), pp.906-919*, 20014.

[Shu16] V. Shukla. Requirements Engineering : A Survey Requirements Engineering : A Survey. (January), 2016.

[Sil14] A. Rodrigues Da Silva. Quality of requirements specifications: a preliminary overview of an automatic validation approach. *29th Annual ACM Symposium on Applied Computing*, pages 1021–1022, 2014.

[Sil17] A. Rodrigues Da Silva. Linguistic patterns and linguistic styles for requirements specification (i): An application case with the rigorous rsl/business-level language. In *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, pages 22:1–22:27, New York, NY, USA, 2017. ACM.

[Sil18] A. Rodrigues Da Silva. Rigorous Requirements Specification with the RSL Language: Focus on Uses Cases. *INESC-ID Technical Report*, 2018.

[SPS18] A. Rodrigues Sa Silva, A. C. R. Paiva, and V. Silva. Towards a Test Speccification Language for Information Systems: Focus on Data Entity and State Machine Tests. *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, 2018.

[SQ15] International Software and Testing Qualifications. ISTQB ® Foundation Level Certified Model-Based Tester Syllabus. 2015.

[SSF$^+$07] A. Rodrigues Da Silva, J. Saraiva, D. Ferreira, R. Silva, and C. Videira. Integration of re and mde paradigms: the projectit approach and tools. *IET Software*, 1(6):294–314, December 2007.

[SSVM15] D. Savic, S. Lazarević S. Vojislav S. Vlajić, A. Rodrigues Da Silva, and M. Milić. Silabmdd - a use case model driven approach. *ICIST 2015 5th International Conference on Information Society and Technology*, 03 2015.

[TPK11] T. Takala T. Pajunen and M. Katara. Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 242–251, 2011.

[UKKD08] E. J. Uusitalo, M. Komssi, M. Kauppinen, and A. M. Davis. Linking requirements and testing in practice. *Proceedings of the 16th IEEE International Requirements Engineering Conference, RE'08*, pages 265–270, 2008.

[WB13] K. Wiegers and J. Beatty. *Software Requirements*. Microsoft, Third edition, 2013.

[WHT17] M. Wynne, A. Hellesoy, and S. Tooke. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, Second edition, 2017.

[Wit07] S. Withall. *Software Requirements Patterns*. Microsoft Press, 2007.

[Yu97] E. S. K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering*, pages 226–235, Jan 1997.

[ZSM17] J. Zander, I. Schieferdecker, and P. J. Mosterman. *Model-based Testing for Embedded Systems*. CRC Press, 2017.