FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Model-Based Testing: From Requirements to Tests

Valter Emanuel Ribeiro da Silva

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Ramada Paiva

July 22, 2017

Model-Based Testing:
From Requirements to Tests

Valter Emanuel Ribeiro da Silva

Mestrado Integrado em Engenharia Informática e Computação

July 22, 2017

Abstract

Automating software testing can significantly reduce the effort, time and cost of software testing throughout the entire development life cycle. Model-Based Testing (MBT) is a software testing technique upon which test cases are generated from a model, an intermediate format requirements document, which provides multiple technical concerns of a given software system. This way it is possible to obtain test cases from requirements models to achieve an automation and systematization of the test process, according to certain coverage criteria.

RSL stands for "Requirements Specification Language", which is a formal language to support and improve the production of system requirements specification (SRS). Developed at Instituto Superior Técnico, Universidade de Lisboa, this approach arranges the different aspects of Requirement Engineering (RE) into several views containing a set of logical constructs. These constructs are defined as linguistic patterns, grammatical rules that guide the production of understandable and coherent textual sentences. Closing the gap of requirements representation and natural language, which is the root of many requirements quality problems (incorrection, inconsistency, incompleteness, and ambiguousness).

This research presents the TSL, acronym for "Testing Specification Language", a model-based testing approach for formal and human-readable specification of test cases that is based on the nomenclature and grammar defined by RSL. By applying Black-Box testing design techniques, TSL allows the construction of three different test patterns, from the perspective of functional system tests, that are expressed in the RSL approach. Namely, Domain Analysis (equivalence partitioning and boundary value analysis for the definition of structural data class values); Use Case Testing (derivation of tests from the various process flows expressed by the use cases); and State Machine Testing (covering the sequence of states from event-based state transitions).

The methodology developed was applied in a case study, a simple fictitious business information system, named "Billing system". This illustrates how TSL supports the testing development cycle as an end-to-end process and the creation of functional tests easy to write, read, validate and maintain.

Resumo

A automação de testes de software reduz significativamente o esforço, o tempo e o custo total do processo de testes ao longo do ciclo de desenvolvimento do produto. *Model-Based Testing* (MBT) é uma técnica de teste de software em que casos de teste são gerados a partir de um modelo, um formato intermédio de requisitos, que fornece vários detalhes técnicos de um determinado sistema de software. Desta forma, é possível obter casos de teste a partir de modelos de requisitos para conseguir uma automação e sistematização do processo de teste, de acordo com os critérios de cobertura definidos.

RSL, termo para "Requirement Specification Language", é uma linguagem formal para apoiar e melhorar a produção e especificação de requisitos de sistema. Desenvolvida no Instituto Superior Técnico da Universidade de Lisboa, esta abordagem organiza diferentes aspectos da Engenharia de Requisitos (RE) em vários níveis através de um conjunto de expressões lógicas. Estas expressões são definidas através de padrões linguísticos, regras gramaticais que mapeiam a produção de frases textuais compreensíveis e coerentes. Desta forma, reduz-se a lacuna entre a representação de requisitos e a linguagem natural, a principal origem de problemas de qualidade dos requisitos (incorreção, inconsistência, incompletude e ambiguidade).

Este trabalho de dissertação apresenta o TSL - "Testing Specification Language", uma abordagem MBT para especificação formal e legível de casos de teste que se baseia na nomenclatura e gramática definida pelo RSL. Pela aplicação de técnicas de design de testes por caixa preta, o TSL permite a construção de três padrões de teste, na perspectiva de testes de sistema, que estão expressos na abordagem RSL. Nomeadamente, Análise de Domínio (criação de classes de equivalência e análise de valores limite para a definição de valores de dados estruturais); testes de casos de uso (derivação de testes a partir dos vários fluxos de evento explícitos pelos casos de uso); e testes de máquinas de estado (extração de sequência de estados a partir de autómatos finitos).

A metodologia desenvolvida foi aplicada num estudo de caso, um sistema fictício e simples de informação empresarial, denominado de "Billing System". Este exemplo permite ilustrar como o TSL suporta o ciclo de desenvolvimento de testes como um processo automizado e a criação de testes funcionais fáceis de escrever, ler, validar e manter.

Acknowledgements

The present work comes as a final result to conclude the degree of Masters of Informatics Engineering and Computing. So I would like to take to this opportunity to express my most sincere gratitude:

To my coordinator Prof. Ana Cristina Ramada Paiva for all the feedback given throughout its development, and for always manifesting open availability to support and direct this project. Also a special thanks to Prof. Alberto Manuel Rodrigues da Silva, for providing his valuable insight and supplying important material for the execution of this work.

To FEUP and all the professors who contributed to my academic education. For providing me knowledge in multiple scientific fields, and most of all instilling in me of what is (in my perception) the true doctrine of Engineering. Discipline, rigor, hard work, humility and analytic view over the functioning of things and the world around us.

To my beloved family, my caring parents Maria and Alberto, my precious little sister Diana and my kind big brother Honorato. For all the sacrifices you've done and for letting me pursue my aspirations even through adversities and difficulties.

To my dear course comrades Teresa, Sara and Daniel (just to mention a few). You've turned this journey into a pleasant experience, making bearable all of those countless nights of work in FEUP.

Lastly, I want to express a very special gratitude to everyone who battles, and still continue to, for an equal and equitable education system. Acknowledging that the right to education and access to knowledge is independent of social status or economics. Whether through public scholarships or accommodation support to underprivileged students, granting everyone the endorsement of an opportunity.

Without these uncountable people this would not be possible.

Valter Silva
Porto, June 2017

"O futuro não nos cai do céu já feito.
É preciso merecê-lo. Se não, é só o futuro dos outros."

Eduardo Lourenço

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

ALM	Application Lifecycle Management
CNL	Controlled Natural Language
DSL	Domain Specification Language
FEUP	Faculdade de Engenharia da Universidade do Porto
GPL	Generic Purpose Language
IDE	Integrated Development Environment
IST	Instituto Superior Técnico da Universida de Lisboa
MBT	Model-Based Testing
NLP	Natural Language Processing
RE	Requirements Engineering
RSL	Requirement Specification Language
SRS	System Requirement Specification
SUT	System Under Test
TSL	Testing Specification Language

Chapter 1

Introduction

This dissertation inserts itself in the domain of software engineering, more specifically within the scope of software testing. Software testing has the main goal of providing stakeholders with information about the quality of the system under test (SUT). This is achievable through the execution of a system, or one of its components, in order to evaluate one or several proprieties of interest. Examining whether the specified requirements are met, the system's inputs assemble the right outputs, the system's efficiency and usability, etc.

This introductory chapter presents the context regarding the problem's domain that is addressed throughout this research work. It is provided a quick overview of its technical background, the major drive for its realization and main goals to be accomplished with this dissertation.

The last section enumerates and briefly describes the chapters that compose this document.

1.1 Context

Nowadays with the exponential increase in software system solutions, and the higher human dependency of software, software quality has become a crucial factor for the success/failure of the final product. When testing one intends to find out if a system's desired and actual behaviors differ or to increase confidence that they do not. Therefore it is is one of the most useful quality checking methods [ULP+06]. The existence of increasingly complex software systems, which are much more time and costing consuming when it comes to testing, makes the practice of manual testing practically impossible. Victor Kuliamin expresses this fact in his work, *Multi-paradigm Models as Source for Automated Test Construction*, by stating:

> "Testing is one of the most advantageous methods of quality checking. But with growth of the software complexity the effort needed to test it throughly seems to grow according to some nonlinear law." [Kul05, chap. Introduction]

Automating software testing can significantly reduce the effort, time and cost of software testing throughout the entire development life cycle. Model-Based Testing is a promising software testing approach upon which test cases are generated from a model, an intermediate format requirements document, which provides multiple technical concerns of a given software system. This way it is possible to obtain test cases from requirements models to achieve, not only, a automation and systematization of the test process, but also contribute to increase the quality of the requirements.

RSL is a system requirement specification language (SRS), developed at Instituto Superior Técnico, Universidade de Lisboa [Rod17]. This requirement model has a formal approach for specification of software requirements that uses lightweight Natural Language Processing (NLP) techniques to translate informal requirements into formal representation. Bridging the gap of natural language requirements representation which is the root of many requirements quality problems (incorrection, inconsistency, incompleteness and ambiguousness).

This research work has explored this testing approach in this new system requirement specification (SRS) language RSL - Requirement Specification Language.

1.2 Motivation and Objectives

During a software development project, testing is one of the most important activities to ensure the quality of a software system. About 30 to 60 percent of the total effort within a project is spent on testing [Ibe13]. It is also estimated that up to 50 percent of the total development costs are related to testing [Fag01]. This denotes not only its importance, but also the higher impact it has in the overall system development process cycle. Model-based testing is one technique that addresses this problem. A potential infinite set of test cases are generated from a test model, an abstract representation of the system to construct. System models vary in nature: textual or graphical; more or less abstract; describing the functionalities of the application under test or test goals; etc. The problem is that often these models do not exist, or there is only a textual description of requirements with an informal structure. The existence of a SRS with controlled natural language, the RSL, enables the derivation test cases directly from a system model.

The aim of this research work is to develop an approach and framework to generate functional tests directly from this formal format of requirements (RSL). So it is expected the following main objectives:

- build a system tests pattern catalog and strategies to construct them over the functional requirements present in RSL language;
- build a tool to specify these detected test patterns and generate test cases directly from a RSL model;
- create an higher level of systematization by automating the test extraction process.

This way it is expected to improve the overall testing cycle process over a RSL model in the following three quality attributes: (1) effectiveness - by reducing the time of test development,

since tests are generated using a model; (2) usability - making easier to produce tests considering the support provided with an editor workspace environment; and (3) correctness - by creating verification and validation over the developed tests.

1.3 Structure of the Dissertation

Besides the introduction this dissertation report contains five more chapters.

Chapter 2, presents the first major concept associated with the state of the art of this research, the RSL Language. It starts by specifying the main problem that drove the creation of this SRS model within the RE domain. Also it is briefly illustrated how it specifies requirements in a formal manner, employing linguistic patterns and linguistic styles, and an overview in its two main levels (Business and System).

Chapter 3 introduces the concept of Model-Based Testing, the testing technique explored in this dissertation. It begins by addressing software testing domain to fully understand this new approach of test case extraction from requirements. Then it presents its definition, fundamental process, limitations and current developed MBT tools.

Chapter 4 describes the conceived solution to apply this technique into a RSL model, the TSL Language. A formal approach to specify system test cases based on the grammar defined by the RSL model. Additionally, it is reported the conception of a support tool to automate the process of graphic visualization and generation of test sequences from RSL state machines.

Chapter 5 provides a case study demonstrating a concrete usage of TSL to specify the different test classes directly from the RSL specification. It is portray a fictitious information system named "Billing System" and specified test cases examples for each of the test class types elaborated.

Chapter 6, presents the final remarks drawn from this research work. Also it is discussed the future work and improvements that can be accomplished.

Chapter 2

RSL - Requirement Specification Language

This chapter presents the first major concept inherent to the state of art of this work, the RSL. RSL is requirement specification language developed at Instituto Superior Técnico de Lisboa. This promising approach comes as a long-term research initiative in the RE field, which attempts to solve the quality problems caused by the use of natural language.

It will start by presenting a very brief notion of Requirements Engineering and how RSL inserts in its domain. Afterwards, it will introduce the RSL language including a quick overview of its two main levels (business and system) and how it defines linguistic patterns and their representation through linguistic styles.

2.1 Requirements Engineering

Requirements Engineering (RE) has the main goal of “establishing a vision within an existing context” [JP93; Poh10], to provide a common understanding of the system under development between the various stakeholders involved. RE encompasses the whole process of elicitation, analysis, documenting/specification, validation and maintenance of requirements. A requirement, in this context, represents a need or a want in the final system. During the last decades, many definitions have been proposed to define what is exactly a "requirement". Ian Sommerville and Pete Sawyer (1997) comprises different visions, expressing that:

> "Requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system." [Wie03, chap. Some interpretations of ”requirement”]

Sets of requirements are used to capture the information needed to design, build and test the software system. Each specified requirement shall be structured to be "distinct, relevant, testable, traceable and unitary, meaning that it addresses one thing and only one thing" [Spa15]. As a result of the requirements development process it is produced by the system requirement specification (SRS), a document agreement among stakeholders that describes a complete knowledge of the system under development (SUT). It contains multiple technical concerns of the system, which includes business requirements and user requirements (e.g. User Stories or Use Cases). SRS are 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 success or failure of a system development is highly dependent on the quality of the SRS produced. Has Karl Wiegers has expressed in his book, *Requirements Engineering*, there are several benefits regarding an high quality requirement process [Wie03]. Which includes:

1. Fewer defects in requirements and in the delivered product.
2. Reduced development rework.
3. Faster development and delivery.
4. Fewer unnecessary and unused features.
5. Lower enhancement costs.
6. Fewer miscommunications.
7. Higher customer and team member satisfaction.
8. Products that do what they're supposed to do.

2.1.1 Difficulties in RE

Planning is one of the most critical steps in software development. When it is not done properly, there are high chances of a project failing by not meeting the stakeholders expectations. Several different studies have been conducted proving that the largest source of software bugs, and consequently system failing, is the requirements specification [IBM06; DC12; RSU+14]. Ron Patton distinguishes in his book, *Software Testing*, four main causes of software bugs: specification, design, code and other types. Demonstrating that specification is around 60% of software bugs source, see Figure 2.1.

Establishing a good and effective communication is key for a successful requirement elicitation process. Customers express their vision of the product to business analysts, in order to define the product vision, scope and expected outputs. Many authors (e.g. Benjamin L. Kovitz in 1998, Harry D. Foster in 2003 and Alan M. David in 2005) had stated that Natural Language is the most used and predominant representation for this exchanging of information, as cited by Alberto

Silva in 2012 [FR; DD12]. However, the usage of natural language for documenting requirements is the cause of the main requirements specifications quality problems, namely: ambiguity, incompleteness, inconsistency and incorrectness [FD12].

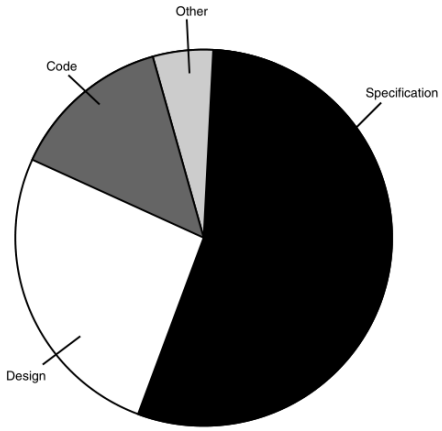


Figure 2.1: Main causes of software bugs. [Pat01, chap. Why Do Bugs Occur?]

2.1.2 The Need for a Domain Specific Language

Since natural language is the main root of requirement quality problems, there is a need of using a more structured and formal language to specify requirements. There are many different approaches to describe data in a machine processable but also human-readable form. For instance, XML (eXtensible Markup Language) and JSON (JavaScript Object Notation) are two widely popular approaches for structured data representation, with many support tools to read, write and parse data according to its syntax. However, specifying a complex and extensive XML specification is too difficult and laborious, and is not very user-friendly to read given the amount of tags [Fow10; BR]. JSON, even though it holds less syntax noise than XML, is still hard to maintain complex specifications [BR]. Both these languages are viable from the perspective of computer processing but not the most suitable for human communication and comprehension, including software requirement modeling. Subsequently, "there was a desire to get the benefits of XML config files without the cost of XML" [Fow10].

A domain-specific language (DSL) is a language used to address a specific application domain, opposite from a general purpose language (GPL) that attempts to solve any kind of software problem. They are expressive languages custom designed for specific tasks, including regular expressions to describe ideas within a scope. Granting a modeling of solutions with an higher level of abstraction, which reduces the complexity degree of the problem within the system context. DSLs can be either textual or graphical. Typically textual DSLs are used for input while the graphical are used for output, as they facilitate the visualization of results.

The Figure A.3 presents three simple example files for describing people, illustrating the primary differences between a XML and DSL specification. The first one is a XML file, the data

is encapsulated between tags which makes difficult to interpret the information. The following version is written in an ad hoc DSL, contains less noise and the data is easier to read. The last one is also a DSL file but even more compact, making the process of read and write more simple and accessible.

```
<people>
  <person>
    <name>James</name>
    <surname>Smith</surname>
    <age>50</age>
  </person>
  <person employed="true">
    <name>John</name>
    <surname>Anderson</surname>
    <age>40</age>
  </person>
</people>
```

```
person {
  name=James
  surname=Smith
  age=50
}
person employed {
  name=John
  surname=Anderson
  age=40
}
```

```
James Smith (50)
John Anderson (40) employed
```

Figure 2.2: XML, ad hoc DSL and DSL example representations, respectively.
[BR, chap. Need for a new language]

2.2 RSL - Requirement Specification Language

RSL is a Domain-Specific Language (DSL), developed at Instituto Superior Técnico da Universidade de Lisboa, designed with the purpose of addressing the requirements elicitation concerns previously mentioned. As described by Alberto Silva (2012), author of the language, "RSL" stands for Requirements Specification Language, emphasizing the purpose of formally specifying requirements. By using controlled natural language, RSL supports the elaboration of "SRSs in a more systematic, rigorous and consistent way" [Rod17].

This language contains a rich set of constructs logically arranged into multiple views according to RE-specific concerns distinguish in two main levels: business level (concepts from the business jargon, e.g. Terminology, Stakeholders and Goals) and system levels (requirements specification of the system, e.g. Architecture, Requirements and Behavior). These formal constructs are "defined as linguistic patterns and represented textually by mandatory and optional fragments" [Rod17]. Although RSL has a formal approach, derived from textual representation, it is human-readable. This way requirements engineers can use RSL directly, or customize to adjust the organization's needs, in order to specify requirements.

This conversion from natural language to formal representation allows several benefits by performing the automation of some tasks, specially related to requirements analysis. In several studies, throughout the development of RSL, it was indicated three major benefits regarding the automation of the RE process [DD12; FD12; FR]. More specifically:

- **Domain Analysis** - identification of all the important concepts, their associations, in order for the system to provide his goals;
- **Verification** - achieving consistency checking on the domain information extracted, through inference and ambiguity resolution;
- **Transformations** - automatically produce different representations for the requirements, such as diagrams, tables, reports, and so on.

Since RSL is a long-term research project, it has adopted throughout its development different formats and frameworks, including a web-based collaborative environment, an Excel spreadsheet and, more recently, an Eclipse and XText-based tool. The last one, since it is the current primary RSL support tool, it is the main study target used in this research. The Eclipse Xtext-based tool is described more in detail later in the methodology segment, Chapter 4.

2.2.1 Definition of RSL Linguistic Patterns and Linguistic Styles

RSL provides a formal specification representation through the use of a controlled natural language (CNL). CNLs define a set of specific terms, arranging them in a restricted grammar (which includes the definition of syntax and semantic of the terms). This lead to a construction of *linguistic patterns*, a set of rules that define both the elements and the vocabulary that shall be used in SRS's sentences. A concrete representation of a linguistic pattern is called *linguistic style* (e.g. RSL linguistic rules).

The following listing 2.1 by Alberto Silva (2017) illustrates the definition of the linguistic pattern and RSL linguistic style for a Stakeholder, a Business level entity responsible for the definition of BusinessGoals/SystemGoals. The Stakeholder pattern rule defines its attributes (specifically: <id>, <name>, <type>, <isA> and <description>). For each one of these elements, is represented the properties name (e.g., id, name, type), typevalue (e.g., ID, String, Integer,Boolean, Enumerated type) and multiplicity (e.g. 1, *, '?'). The multiplicity of an attribute is by default "1" (mandatory single value), or can be represented by the following characters '?', '+', and '*' meaning, respectively, 0..1 (optional), 1..* (more than one), and 0..* (more than zero).

The type of an attribute can be: a predefined type (e.g., ID, String, Boolean); an element type (e.g., the Stakeholder of the isA attribute); or a vocabulary type (e.g., the StakeholderType of the type attribute).

StakeholderType vocabulary rule is prefixed with the "enum" tag and defines the values of its parts, namely the literals 'Organization', 'OrganizationalUnit', 'Team', 'Person', 'System' and 'Other'.

Listing 2.1: Linguistic pattern and RSL style example - Stakeholder [Rod17]

```
// Linguistic Stakeholder Pattern
Stakeholder::
<id:ID> <name:String> <type:StakeholderType> <isA:Stakeholder>? <description:
    String>?
enum StakeholderType::
Organization | OrganizationalUnit | Team | Person | System | Other;

// Linguistic Stakeholder RSL Style
Stakeholder:
        'stakeholder' name=ID ':' type=StakeholderType '{'
                ('name' nameAlias=STRING)
                ('category' category= StakeholderCategory)
                ('isA' super=[Stakeholder])? &
                ('partOf' partOf=[Stakeholder])? &
                ('description' description=STRING)?
        '}';
```

A concrete valid example of the final application of the RSL Stakeholder linguistic pattern is presented on the listing 2.2. The adoption of such a standard format, allows a concise, clear and simple representation. The sentences produced should be short, simple and with limited vocabulary terms. As demonstrated on the next example.

Listing 2.2: Concrete RSL style represention example - Stakeholder [Rod17]

```
// RSL Stakeholder representation example
stakeholder stk_user : Person {
        name "User"
        category Business_User_Direct
        isA stk_user
        description "User of the system"
}
```

The adoption of CNLs show the following advantages: (i) the CNL sentences are easy to understand, since they are similar to sentences in NL; (ii) are less ambiguous than expressions in NL, since they have a simplified grammar and a predefined vocabulary with a precise semantics; and (iii) are semantically verifiable and computationally manipulated, since they have a formal grammar and the predefined terms.

Linguistics patterns are grammatical rules that can be stated simultaneously in a descriptive and prescriptive manner, and that allows their users to properly speak in a common language. From the linguist's point of view, a grammar is not only a collection of rules, but rather a set of blueprints that guide speakers in producing comprehensible and predictable sentences.

2.2.2 RSL Language Overview

As described previously, RSL contains several constructs logically arranged in different viewpoints according to various RE aspects addressed, namely: stakeholders, goals, scenarios, qualities and constraints, rationale and assumptions, definitions, measurements and priorities. These viewpoints are organized in two main abstraction levels: business level and system level. The following scheme (Figure 2.3), presented in *Linguistic Patterns and Styles for Requirements Specification: The RSL/Business-Level Language* (Alberto Silva, 2017), illustrates these two main abstraction levels comprising the various RE concerns within its addressing interrogatives.

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

Figure 2.3: Classification of RSL viewpoints: abstraction levels versus RE specific concerns. [Rod17]

The RSL Business Level - this first level provides an understanding over the business context of the system. This is comprised in the RSL business level with the following viewpoints: Terminology, Stakeholders and Goals. The Terminology view defines the project's business nomenclature, containing the terms regarding the business jargon. The Stakeholders view arranges the people or organizations that are involved with the system-of-interest. For last, the Goals view holds the objectives projected from the various stakeholders for the expected and final system.

The RSL System Level - with the business-related concerns specified, RSL then addresses the system's problem domain, of both static and dynamic elements, to specify in detail its requirements. These aspects are comprised in the RSL system level with the following viewpoints: Architectural; Requirements; Structural; and Behavioral. The Architectural view encompasses the system's definition and sub-systems that compose it, and their relation. On the other, the Requirements view documents the desired features to be implemented and non-functional quality attributes. The Structural view designates the data entities and business entities with their specific attributes, characterizing the whole system's structure. Finally, the Behavioral view blueprints the construction of functional aspects of the system by further detailing the actors who interact with

it, the event-based state machines and the use cases.

2.3 Conclusion

This chapter introduced the formal requirement language "RSL", namely its insertion in the RE domain and the issues it attempts to solve, its definition, the design of linguistic patterns and linguistic styles (exemplified with the representation of a Stakeholder) and its two main overview levels (the business and system levels).

Analyzing the RSL formal architecture to handle specification of requirements, it is clear the advantages and opportunities of this promising approach. By allowing the automation of the requirement validation process it warrants better requirements quality, reducing the main roots of requirements problems previously mentioned. Also, since it is a formal representation, besides allowing a computer-processable manner of information extracted by natural language requirements, it also is human understandable. Which can be a great interlingua for communicating requirements between stakeholders and developers.

Chapter 3

Model-Based Testing

In this chapter it is described the second main concept behind the state of art of this project - Model-Based Testing (MBT).

Firstly the main concepts of software testing are introduced to fully understand this new approach of test case extraction: the importance of software testing to assure software quality, the fundamental testing process and different test levels throughout the software development cycle, test design techniques (white-box and black-box), and test generation from requirements approaches (in which MBT is included).

Thereafter, it will be expressed in detail the notion of Model-Based testing, its definition and main general process, some MBT limitations/misexpectations and most notable developed tools. Demonstrating the potential of this approach towards the RSL specification language.

3.1 Software Testing

Software systems are an intrinsic part of everyday life, from web and mobile applications (e.g. social networks) to highly critical systems (e.g. banking applications). This modern reality leads to bigger concerns when it comes to the effective quality and reliability of software control techniques. The existence of human mistakes (errors) produces into the system's code bugs/defects. When these bugs are executed the system fails, not returning the expected result (failure). Also, failures can be generated by running the system in extreme environmental conditions.

Testing has the main role of reducing the risk of problems occurring during operation, which implies the consolidation of software quality [IST15]. There are different approaches when it comes to defining software quality. This includes, the degree to which a system, component, or process meets the specified requirements. Also, the IEEE Standard of Software Engineering Terminology Glossary, defines it as "the degree to which a system, component, or process meets customer or user needs or expectations" [IEE90]. As expressed by the International Organization for Standardization (ISO) it is "the totality of characteristics of an entity that bear on its ability

to satisfy stated and implied needs" [Iso94]. As expressed on the last chapter, requirements specifications are equally held of inherent quality problems meaning that meeting the requirements specifications not always implies product quality.

Covering this last issue, ISO defined IEC 9126 standard [ISO00] which distinguishes software product quality in three views:

- **Internal Quality -** is the totality of characteristics of the software product from an internal view during its development or maintenance (e.g., code, architecture)
- **External Quality -** is the totality of characteristics of the software product from an external view during its execution
- **Quality in Use -** is the user's view of the quality of the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself (e.g., usability)

The primary goal of software testing is "to find "bugs", find them as early as possible, and make sure that they get fixed" [Pat01]. This allows an higher degree of confidence on the software correctness, which ultimately assesses to software quality.

3.1.1 Fundamental Testing Process

During the software's life cycle models there are various development phases. This includes the requirement's elicitation, software design, implementation (coding), testing and deployment. For each one of the development phases it is necessary to proceed different types of testing, which leads to different test levels. Ilene Burnstein acknowledges in her book, *Practical Software Testing: A Process-Oriented Approach*, the main test levels with an extended version of the V-model, see figure 3.1. These test levels are respectively: Unit Testing, Integration Testing, System Testing and Acceptance Testing.

- **Unit Testing -** Testing of individual software/hardware units, or groups of similar units. It is usually done by the developers to make sure that their code is working correctly. They test their piece of code in order to detect functional and structural defects.
- **Integration Testing** - Testing when two modules, or components, are combined and evaluates the interaction between them. It has the main goal of assuring the correct behavior and functionality of both modules after their integration.
- **System Testing -** Testing executed on the complete, integrated system to verify that it meets with the specified requirements. Usually by the responsibility of an independent test team, and evaluates the system performance according to his specification, both functional and non-functional requirements.

- **Acceptance Testing -** Testing conducted with the customers, or other stakeholders involved. It determines whether, or not, the system complies with the acceptance criteria along with the final users of the system. This allows to determine if the customer requirements and expectations are met.

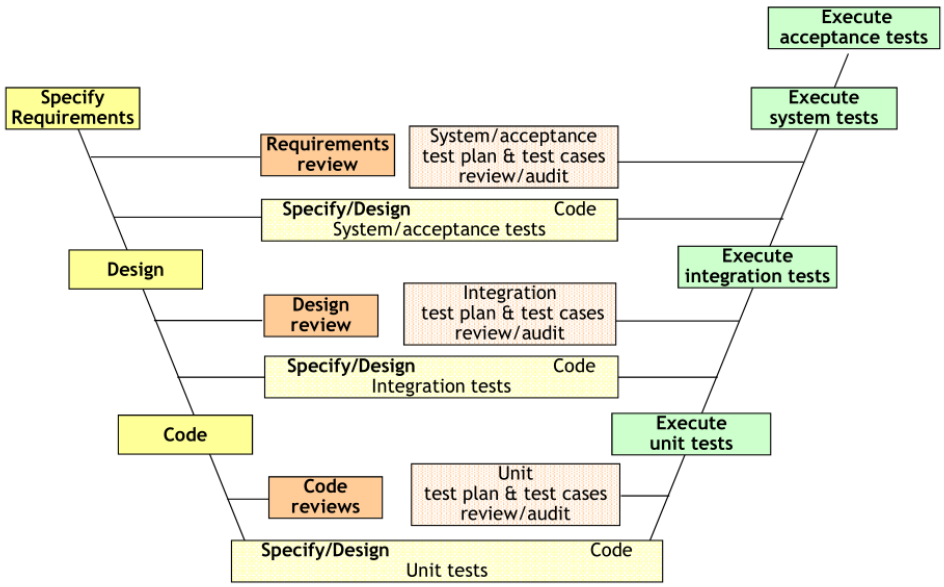


Figure 3.1: The extended V-model of software development by Ilene Burnstein [Bur02]

3.1.2 Test Design Techniques

Test design techniques are used to obtain a set of test cases (test suite) that accomplish a certain specified coverage criteria. It starts with the specification of test situations, derived from the application of various coverage types. The elaboration of test cases is an expensive task, as expressed by Dijkstra:

> "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra, 1972]

This denotes the fact that since exhaustive testing is usually impossible, it is necessary to define a series of test strategies. Since we can not test every single possibility within the SUT, it is necessary to write and execute the fewest number of test cases who can detect as many defects as possible. To achieve this, there are a group of test design techniques that contribute to the reduction of writing redundant test cases, which have similar conditions to other tests already wrote. Given this view there are two basic strategies that can be used to design test cases. These are called the black box (also mentioned as functional or specification testing) and white box (sometimes called clear or glassbox) test strategies. These approaches are summarized on the following Figure 3.2

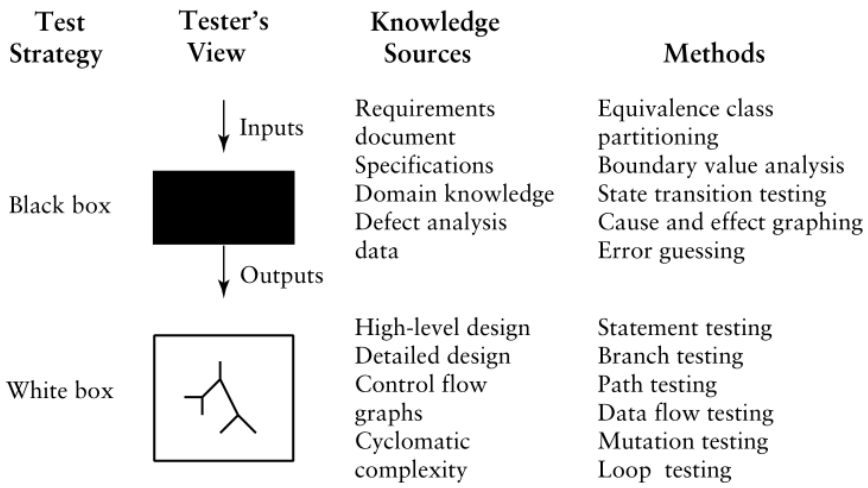


Figure 3.2: The two main basic testing design strategies by Ilene Burnstein [Bur02]

3.1.2.1 White-Box Testing

White-box testing, also mentioned as structured-based testing and glassbox testing, is a software design technique that evaluates the code and internal structure of the program. This approach takes insight of the internal perspective of the SUT and derivate test cases according to its program structure. This testing technique is used by both developers and testers. It helps them understand which line of code is actually executed and which is not. This may indicate that there is either missing logic or a typo, which eventually can lead to some negative consequences. White-box software testing techniques can be applied mainly at the unit, integration and system levels of the software testing process. There are several White-box test techniques following different code coverage criteria, including:

Statement testing - Test technique performed based on the statement coverage, the total number of code lines executed (covered). The coverage percentage is obtained by dividing the number of statements that have been executed, with the specified test cases, by the total number of all executable statements of the code under test. Through statement coverage, every line of code needs to be checked and executed, allowing the detection of code sections that are never reached.

Decision Testing - White-box technique, also mentioned as branch testing, which the specified test cases tries to cover all of the decision points (e.g., Boolean expressions like the IF statement) of the program under test. The decision coverage percentage is calculated by the number of all the decision points executed by the number of all possible of decision outcomes that the program under test contains.

Path Testing - Methodical testing approach in which test cases are designed to exercised every possible path in the program. A linear independent path is defined following the control flow graph of the application. This includes all decision points and loop paths taken to zero to maximum (ideally). In this type of testing every statement in the program is guaranteed to be executed at least once.

Data flow testing - Test strategy based on the selection of paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects. Data flow testing focuses on the points at which variables receive values and the points at which these values are used.

Mutation testing - Modern test approach in which faults (or mutations) are automatically inserted into the code, modifying small parts of the program. The test cases are executed in order to detect and reject mutants, this way the mutant is killed. If the tests pass, not detecting the mutant, then the mutation lived. This allows to evaluate the quality of the software system under test by locate weaknesses in the test data used for the program or to discover unreachable statements.

3.1.2.2 Black-Box Testing

Black-box testing, also denominated as behavioral testing or functional testing, focuses on determining whether (or not) a program does what it is supposed to do based on its functional requirements. Through this testing it is possible to detect errors in the external behavior of the system, independently of the code, by several categories: (1) incorrect or missing functionality; (2) interface errors; (3) errors in data structures used by interfaces; (4) behavior or performance errors; (5) initialization and termination errors.

Usually this testing technique is not done by the programmers of the code but by independent testing groups, often third-party testers. This way it is possible to make sure that the system does what the customer wants it to do. Testers should be able to understand and specify, for a given input of the program, what the desired output should be.

The elaboration of test cases is an expensive task. Since it is impossible to test every single possibility within the SUT, it is necessary to write and execute the fewest number of test cases who can detect as many defects as possible. To achieve this, there are a group of black-box strategies that contribute to the reduction of redundant test cases. Below it is presented a set of black-box methodologies:

Equivalence class partitioning (ECP) - This is a software testing technique that divides the input data of the application under test into partitions of equivalent data from which test cases can be derived. By grouping test cases with similar characteristics it is possible to reduce the number of test executed, uncovering classes of errors. This also allows a reduction of testing processing time since there are fewer number of test cases.

Boundary value analysis - Software testing technique in which tests are designed to include representatives of boundary values in a range. The idea comes from the concept of boundary, since it requires an analysis of the frontier values of the equivalence partitioned classes (values immediately above and below each of established class). This method is also consider a complement towards the partitioning method in equivalence classes.

State Machine testing - Tests cases are designed to execute valid and invalid sequences of transactions from state machines. A state machine is a model that describes the dynamic system behavior. Outputs are triggered by changes to the input condition, causing a change of the system state. This is represented in a state transition graph, in which the nodes represent the states and the arrows represent the transitions, that occur after an input or event.

Cause Effect Graph - Technique in which a graph is constructed by mapping a set of causes (the inputs) and the respective effects (the outputs). This method is useful when there is a wide amount of testing conditions which can't be executed separately. The graph generated from this method contains on the left side nodes representing the causes, on the right side the nodes represent the effects. Later it can be converted into a decision table, from where the test cases will be generated.

Error Guessing - Test method in which the test cases designed are established based on prior testing experience. This relies mostly on the software tester involved, including his past experience in order to find causes for software failures. Error guessing has no explicit rules for testing, test cases are conceived based on intuition depending on each situation.

Use Case Testing - Testing technique where test cases are extracted from use cases. A use case is a description of a particular use of the system by an actor (a user of the system). They help define the functional system software requirements by illustrating a process flow of the actual real use of the system. For each use case there is, usually, at least one basic scenario (or main scenario) and zero or more alternative/exceptional flows. This way it's possible to exercise the whole system on a transaction basis.

3.1.3 Test Generation from Requirements

The test process, for higher effectiveness, should start as early as possible in the software development life cycle. This way defects can be found in early stages, such as requirements or design phases, since it is much cheaper to fix them beforehand. Errors in requirements specification or architecture tend to cost 50 to 200 times as much if corrected late in the project life cycle than if fixed closer to the origin point [McC09]. This has become a standard guideline for testing, over the past 40 years, and specified as one of the 7 principles of software testing.

> **"Principle 3 - Early testing**
>
> To find defects early, testing activities shall be started as early as possible in the software system or system development life cycle, and shall be focused on defined objectives." [IG11, chap. Seven Testing Principles]

This previous issue can be easily addressed by generating test cases from requirements specification. "Requirements serve as the starting point for the generation of tests" [Mat08]. They contain multiple technical and non-functional details that allow to express how the SUT should, or not, behave. The requirements specifications are divided in two forms: informal/rigorous and formal. Initially these requirements are merely ideas, or informal specifications, that express the intentions for the final product. Rigorously specified requirements can be conceived by modeling elements such as use cases, state machines, UML class diagrams, etc. Finally, for higher effectiveness, this format can be transformed into formal requirements by applying a formal language (e.g. RSL as described in Chapter 2.2). The more formal the specification, the greater the degree of automation for the generation of tests.

The Figure 3.3 presents a set of techniques, described by Aditya P. Mathur in the book *Foundations of Software Testing*, for generating test cases from different formats of requirements.

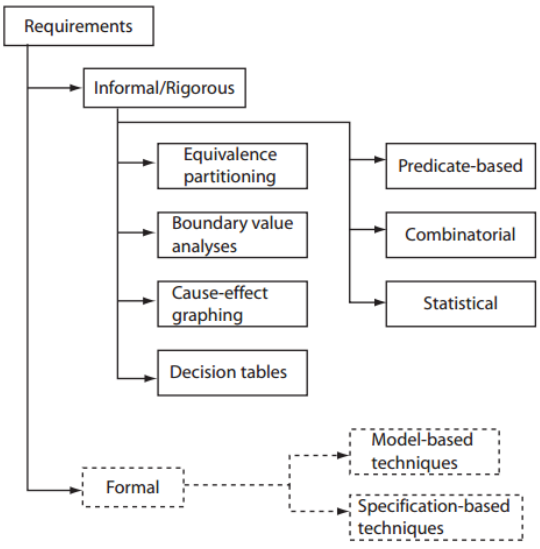


Figure 3.3: Example of techniques for test selection from informal/rigorous and formal specified requirements. [Mat08, chap. Test Generation from Requirements]

In informal/rigorous requirement formats, test cases can be derived by applying directly black-box testing techniques mentioned previously, Chapter 3.1.2.2. This set of techniques has the main goal of reducing the total number of test cases by selecting an input domain, identifying similar input classes for the SUT.

In model-based testing it is produced a test suite on the basis of a specification. The existence of a formal specification or model introduces the possibility of automating test generation, allowing a more efficient and effective test generation process. Usually we aim to produce a test suite that satisfies a given property, a test criterion. The test criterion is typically associated to some notion of coverage.

In the next section it is described in depth the Model-Based Testing approach, which is the main methodology related to this research work.

3.2 Model-Based Testing

Model-based testing is a software testing technique in which test cases are generated, whole or in part, from a model that describes some aspects (usually functional) of the System Under Test (SUT). Models are abstract representations of systems. They help to specify, understand and predict the system's behavior in many disciplines. Models can be applied in many ways throughout the whole product life-cycle, this includes: requirements specification, code generation, reliability analysis and test generation [AD97]. They are an economical ways of preserving the knowledge of the system and then reuse it as the system grows, proving to be a major advantage for software production.

MBT takes advantage of this useful information by deriving and executing test cases from the SUT, typically as black-box perspective. Therefore, MBT not only allows the generation of test cases, but may also act as an oracle. An oracle, in software testing engineering, is a mechanism for determining whether a test has passed or failed. MBT also supports and extends classic test design techniques, mentioned previously, such as: equivalence partitioning, boundary value analysis, decision table testing, state transition testing, and use case testing.

As described by the Model-Based Tester Syllabus [IST15], the main idea is to improve the quality and efficiency of the test design and test implementation activities by:

1. Designing a comprehensive MBT model based on project test objectives.

2. Providing a model as a test design specification. This model includes a high degree of formal and detailed information that is sufficient to automatically generate the test cases directly from it. MBT and its artifacts are closely integrated with the processes of the organization as well as with the methods, technical environments, tools, and any specific life-cycle processes.

This way, MBT helps to systematize and automate more of the testing process. The test cases are generated according to coverage criteria. Since the models vary in nature, textual or graphical, more or less abstract, the techniques applied to generate test cases depend in such nature of the given model.

3.2.1 MBT fundamental process

The fundamental MBT process, as expressed by Mark Utting, encompasses five major steps (Figure 3.4). Applying the MBT approach in the testing software life cycle will, at the very least, impact the activities of test analysis and test design [ULP+06]. The following described process should be executed iteratively and incrementally [IST15].

Step 1. Model conception of the SUT from the informal requirements or existing specifications. This model, also known as test model, describes functional aspects of the system. This way test models are an abstract representation of the system, which implies that they are simpler and easier to maintain than the actual SUT.

Step 2. Test planning by the definition of test selection criteria and metrics. This allows the formal representation of test case specifications, guiding the mapping of feasible test suites.

Step 3. Test design, formalization of test cases by applying the previously test selection criteria. Test case specifications are an higher level description of the test case, being an operational representation of the selected test criteria.

Step 4. Test generation, through production of a test suite that satisfies the test case specification. Usually, there are many sets of test cases that fulfill it, so it is important to select one that reduces the testing effort. This way it is possible to minimize the test suite, so that a smaller amount of generated test cases it is covered a large number of test case specifications.

Step 5. Test execution by running the generated test cases. This process can be done manually (e.g. tester) or automated by a test execution environment, through automatic running of the tests and recording the results obtained.

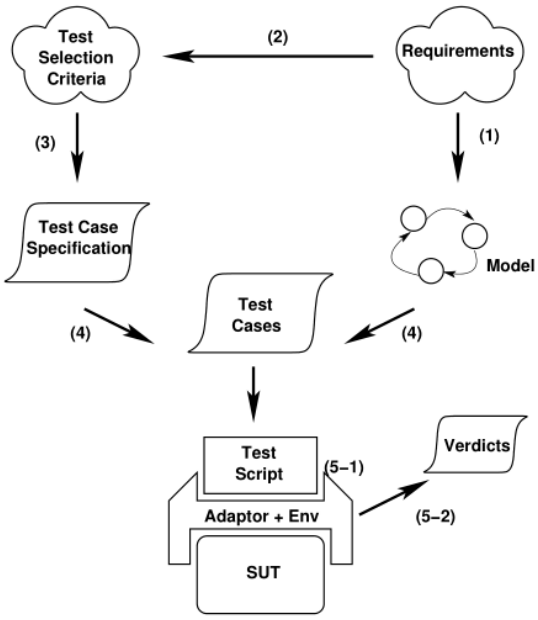


Figure 3.4: Fundamental Model-Based Testing process by Mark Utting [ULP+06]

3.2.2 Limitations and Misleadings with MBT

Model-Based testing presents a promising approach to the quality checking of software systems, supporting test automation through the usage of test models. But regarding the benefits and advantages of this technique, mention previously, does not lead to a straightforward utility and practicality in real situations. Several studies [IST15; Kul05] presented some problems and limitations to this technique that need to be taken into account, including:

MBT does not replace the full testing life cycle process. MBT extends previously mentioned classic test design techniques, Chapter 3.1.2, allowing an efficient extraction of test cases from developed test models. However, since models are a simpler abstract representation of the SUT, some possible variant behaviors may not be explicit. For instance, in complex systems it's very difficult to map the full behavior into the model, which may lead to the absence of possible system's details.

MBT is not just a matter of tooling. Although MBT takes support of automation frameworks, it impacts and changes the whole testing process. So it is still important to establish objectives and quality assessment metrics.

Models are not always correct. As in conventional testing, errors can be introduced into the test model leading to wrong test case results. Also, software systems specifications are not static artifacts, given the constant change of the requirements this needs constant adaptability. If the test model is not adjusted during test development, the test artifacts generated can become obsolete.

Test case explosion. When applied a combinatorial test case execution into the defined test cover criteria, this may lead to an extensive number of obtained test cases (test case explosion). This issue can be fixed by applying filter mechanisms into the test case generation algorithm.

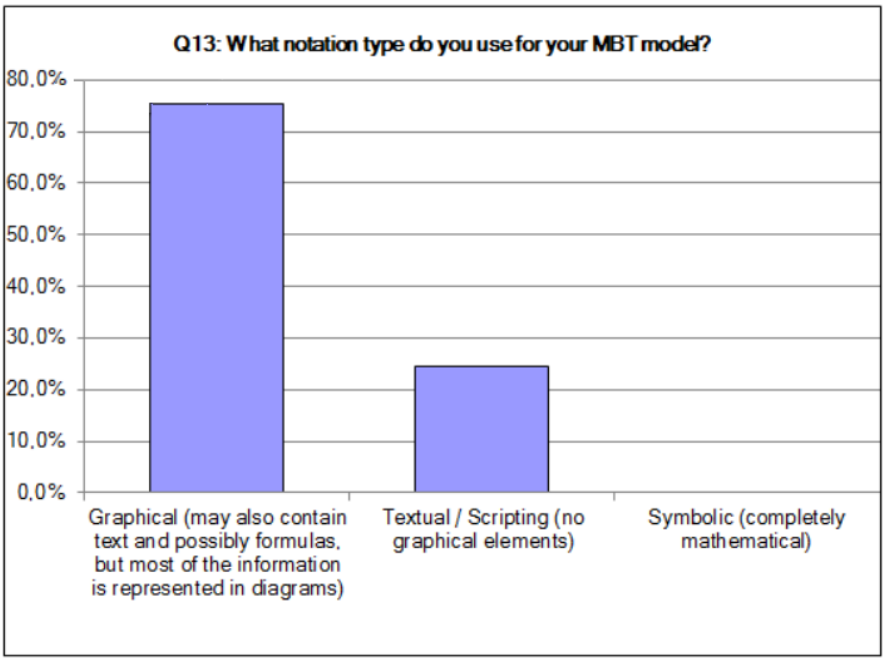
3.2.3 MBT Tools

During the last three decades there has been significant research in the MBT field. But only in the past few years, the area of model-based testing has evolved from an academic research topic to an emerging practice in the industry, with growing development of MBT commercial and open-source solutions. Recent emphasis in test focused methodologies, such as Behavior Driven Development, and the technological progression in software testing lead to an huge increase of MBT approaches, consequently to actual tools.

Since the MBT tools are highly dependent on the test model, which can diverge from graphical (e.g. UML) to specification based (e.g. RSL), the scope, characteristics, test selection criteria and output artifacts vary significantly from different approaches. A taxonomy model was developed in order to categorize MBT tools into different criteria, namely: model specification, test generation and test criteria [ULP+06].

Also many survey studies in the field of model-based testing provide a succinct overview to the current MBT practices. The following table is comprised of some of the most notorious MBT tools developed over the last few years. It is illustrated in some detail the year, the input format types and a brief tool description. It is noticeable that most of these developed tools are based

from a graphical notation, since they comprise a more formal structure making easier to extract test cases. In comparison textual based models are less adopted in MBT approaches given that they are mainly expressed through informal natural language, as mentioned in Chapter 2.1.1. In the graphic below is expressed the results from the "2016/2017 Model-based Testing User Survey" [KBB+17], indicating that only around 25% of the MBT approaches are based from textual/scripting elements, against 75% of graphic-based MBT solutions. Also, it is observable an increasing bet regarding textual MBT solutions, attending to the 14% of this type of solutions registered on the preceding survey realized in 2014.



Answer Options	2016	2014
Graphical	75,5%	81,0%
Textual / Scripting	24,5%	13,9%
Symbolic	0,0%	5,1%

Figure 3.5: Notation type used for the MBT model - 2016/2017 Model-based Testing User Survey: Results [KBB+17]

Table 3.1: Overview list of selected MBT tools based on Zoltán Micskei's survey study [Mic16]

Tool/Ref.	Year/Type	Input Format	Description
4Test	2016 Commercial	Custom (Gherkin based)	4Test uses a combinatorial approach called constraint driven testing to select test cases from textual models specified in a syntax inspired by the Gherkin language. *Abstract Data Type*
Conformiq Creator	2016 Commercial	Activity Diagrams, DSL	Creator allows the creation of models via import from existing assets (e.g., flowcharts, BPM, Gherkin and manual tests) and export generated tests to ALM tools, Excel, various scripting languages, or test execution with Conformiq Transformer.
GraphWalker	2014 Open Source	FSM	Test generation from Finite State Machines. Search algoritms: A* or random, with a limit for various coverage criteria (state, edge, requirement). Formerly called as mbt.
fMBT	2014 Open Source	Custom (AAL)	fMBT (free Model-Based Testing) generates test cases from models written in the AAL/Python pre/postcondition language using different heuristics (random, weighted random, lookahead...).
JTorx	2014 Open Source	LTS	JTorX is a reimplementation of TorX in Java with additional features. The LTS specification can be given in multiple format, and it can interact on-the-fly with the implementation under test.
MBTsuite	2016 Commercial	UML or BPMN	MBTsuite can generate test cases from UML models based on various coverage criteria (path, edge...) or randomly. It can import model from several model editors and it is integrated with various test management and test execution tools.
ModelJUnit	2014 Open Source	EFSM	ModelJUnit allows you to write simple finite state machine (FSM) models or extended finite state machine (EFSM) models as Java classes, then generate tests from those models and measure various model coverage metrics.
Spec Explorer	2013 Commercial	C# program models	Spec Explorer is a MBT tool that extends Visual Studio allowing to model software behavior, analyze the behavior, check the model and generate standalone test code from them.
Tcases	2015 Open Source	Custom	Tcases is a combinatorial testing tool where the inputs of the system could be specified in an XML file (with conditions, failure values, don't cares, etc.). Tcases can generate n-wise or randomized test suites.
PGBT	2016 Academic	Paradigm	Pattern-Based Graphical User Interface (PBGT) method for systematize and automate the GUI testing process, by sampling the input space using UI Test Patterns that express recurrent systems' behavior [MPN+17].

3.3 Conclusion

This chapter presented the technique explored in this research - Model-Based Testing. It started by presenting the concepts inherent to software testing, to fully contextualize the MBT approach. More concretely, the fundamental testing process with the extended V-model, test design techniques (both black-box and white-box) and how test cases are generated from requirement specifications. Then this approach it is explained with the description of its fundamental process, some misexpectations and limitations and an overview regarding current MBT tools.

MBT, by checking the conformity between the implementation and the model of the SUT, introduces more systematization and automation into the testing process. The potentialities and advantages of model-based testing are extensive. The changes produced over the existent testing process, promote the reduction of the time and costs spent throughout the whole system life-cycle. Which ultimately lead to an increase in software quality.

Regarding this benefits, it is conclusive the enormous potential of RSL language in a business environment. The generation of functional system tests by Model-Based testing, could reinforce the use of this SRS in a professional business environment.

Chapter 4

TSL - Test Specification Language

With the State-of-Art associated with this research presented, the software requirement specification (SRS) language RSL and the model-based testing approach (MBT), now it is described the conceived solution to apply such technique into this model.

TSL, acronym for "Testing Specification Language", is a model-based testing approach to specify test cases based on the grammar defined by the RSL Xtext model. In this chapter this solution is described in depth. It starts by describing the used tools (XText framework and Eclipse work environment), an overview to this solution and how it constructs functional test patterns based on the expressed grammar in the RSL approach.

Also, to provide an higher degree of automation to the final solution, it was created a TSL State Machine Support Tool, allowing graphic visualization and obtainment of state sequences from RSL state machine specifications.

Finally, although the RSL Xtext model is the most rigorous and completed version, it is also presented an extension of the RSL Excel format to support the TSL approach.

4.1 Modeling Functional Tests through a Domain-Specific Language

Conventionally manual system tests, in the same way as requirements, are written in natural language. The resultant test cases are ineffective since they are hard to write and costly to maintain. Leveraging DSLs for functional testing can provide several benefits regarding good software practices. Robin Buuren acknowledges in his work, *Domain-Specific Language Testing Framework*, three major quality aspects concerning the use of DSLs for testing specification [Buu15], namely:

1. **Effectiveness:** reduces the time of test development, since tests are generated using a model;

2. **Usability:** it is easier to produce considering the support provided by the work development environment IDE;

3. **Correctness:** make system tests clearer by giving testers a programmatic and strictly defined rules, leading to fewer bugs;

Given the formality of the RSL approach to model requirements, system test cases from a Black-Box perspective can be easily generated maintaining the same systematic and rigorous way. This lead to a creation of a new domain-specific language that addresses the testing domain issues from a RSL model. This new language was named TSL, short-term for "Testing Specification Language".

Before fully addressing the TSL language and how test cases are obtained from a RSL model, it is important to understand how DSLs are implemented and the tool in which was implemented, the Xtext Framework.

4.1.1 Implementing a DSL

The implementation of a DSL comprises several parts in order to obtain a distinct jargon that addresses a certain domain problem. A program should be able to read the DSL text, parse it, validate it and may even generate external artifacts from it. Regardless of the implementation method, a DSL encompasses the following phases:

1. **Abstract Syntax:** defines a tree representation of the language syntactic structure according to its primitive elements, and describes how those elements are combined. The abstract syntax is independent of any particular language encoding or representation.

2. **Concrete Syntax:** also known as parse tree, is an ordered, rooted tree that describes the specific representations of the language, including its encoding and/or visual appearance.

3. **Semantics:** process of relating syntactic structures (e.g., phrases, clauses, sentences) to the level of the writing as a whole, to their language-independent meanings.

Martin Fowler, one of the major proponents of the DSL concept, classifies a DSL in two main categories: internal or external [Fow10]. Internal DSLs, also known as embedded DSLs, are languages encapsulated within a general purpose language (GPL). Consequently an internal DSL is determined over the syntax of its host language, incorporating the same tools and editor IDE. Meanwhile, External DSLs are standalone languages, constructed to be parsed independently from other languages. In most cases these are more difficult to implement than internal DSLs, since it is necessary to define tools for its editing, checking, parsing and execution. As an example, regular expressions or Cascading Style Sheet (CSS) are two representations of external DSLs.

Xtext Framework supports the development of external DSLs. The next section introduces this tool and how it constructs a fully domain specific language and how it integrates with the Eclipse IDE editor.

4.1.2 Xtext Framework

XText is an open source framework for implementing DSLs with the integrated development environment (IDE) Eclipse. It allows a quick language development, supporting all important aspects that covers a complete language infrastructure and incorporating them into a complete Eclipse IDE, taking advantage of its provided features. It grants a quick implementation of all the components previously mentioned by automatically handling their construction from a grammar specification. By elaborating a Xtext grammar it is possible to generate the lexer, the parser, the AST model (which represents the parsed program) and create a concrete DSL representation from the Eclipse IDE editor.

A Xtext grammar (defined in a .xtext file) is specified using an EBNF-like notation, containing a collection of parser rules. These rules specify the concrete syntax of the language, also let its mapping to the Abstract Syntax. Each syntax rule confines a structure from the domain language that through a set of Tokens (e.g., ID rule, string value, integer value, reference value to other defined entity, special characters, etc.).

When selecting a new project with Eclipse IDE the Xtext wizard generates multiple folders (with a name based on the Project name) comprising different aspects from the language. Considering the TSL project, we have the five sub-project folders:

- **org.itlingo.tsl:** main project folder, where the grammar and the runtime and generate modules are defined;
- **org.itlingo.tsl.ide:** folder comprising the details from the user interface that are independent from Eclipse. Useful for external tool integration, such as IntelliJ and web integration.
- **org.itlingo.tsl.tests:** definition of JUnit tests dependent from the main folder (org.itling.tsl).
- **org.itlingo.tsl.ui.tests:** definition of JUnit tests dependent from the user interface folder (org.itlingo.tsl.ide).
- **org.itlingo.tsl.ui:** contains the components related to the Eclipse UI, which encompasses the Eclipse editor and features from the Eclipse tooling.

After the grammar is complete, it is possible to run the Xtext MWE2 (Modeling Workflow Engine 2) DSL to configure the generation of its artifacts. During the MWE2 workflow execution, Xtext will generate artifacts related to the UI editor of the DSL, and it will also derive an ANTLR specification from the Xtext grammar with all the actions to create the AST while parsing. The classes for the nodes of the AST will be generated using the EMF framework. The generated code is placed in the source folder src-gen, existent in each of the project's sub-folders. After the construction of the grammar and its components (parser and AST) is completed, it is now possible to initiate a new Eclipse instance and create DSL specifications, in this case TSL.

In the next section are outlined some key aspects concerning the Eclipse Editor Environment for the composition and validation of TSL files.

4.1.3 Eclipse Editor Environment

The integration with an established and powerful IDE as Eclipse it makes possible to take advantage of its provided features. Lorenzo Bettini, enumerates in the book *Implementing Domain-Specific Languages with Xtext and Xtend* [BR] a series of relevant features regarding the use of the Eclipse IDE to compose a DSL. More specifically:

- **Syntax Coloring -** Highlighting the various language elements (e.g. keywords, variables, strings, etc.) through color and format using a visual style. Aside from the cosmetic effect, this allows a more effective comprehension and quicker detection of syntactic errors of the program. Which ultimately leads to an higher correctness.

- **Background validation -** The Eclipse IDE provides a real time compilation of the written program, displaying instantly encountered errors. By continuously running background checking over the written program, even if the program is not saved, errors are detected sooner with less fixing costs.

- **Error markers -** The errors detected by the compiler are directly underlined, typically in red, only in the parts of the program that doesn't match the grammar definition. It also supports error markers with an explicit message and fills the Problems view with the correspondent errors. This grants a focal point for the programmer to easily spot the error's source to be fixed.

- **Content assist -** This feature allows an automatic suggestion on how to complete a statement/expression in the program. While writing a program by using the Eclipse command *Ctrl + Space Bar* it is given a list of possible solutions for the current sentence. This provides an higher level of productivity, since the programmer doesn't have to forcedly know all the DSL syntax.

- **Hyperlinking -** Eclipse IDE supports automatic link references throughout the program. For instance, it's possible to navigate directly to a variable declaration, from one of its instances, or to a function definition, from a function call. This procedure, in eclipse, is done by the keyboard key *F3* or using *Ctrl + click*.

- **Quick fixes -** The Eclipse editor provides some correct suggestions when errors are written, typically displayed by a context menu from the error marker. As an example, if a programmer invokes a method that does not exist, it'll be shown a quick fix to create automatically a new correspondent method.

- **Outline -** While programming, the Eclipse IDE contains an outline navigation menu displaying the main program components written. This side menu integrates the elements with their hyperlinks, which permits through a click direct access to the corresponding source line in the editor.

The Figure 4.1 illustrates an instance from Eclipse IDE to create and specify a TSL file. This workspace comprises these previous features into different UI components: (a) Project explorer - a navigable menu containing the multiple projects created, their sub-folders and files; (b) Outline - menu displaying the main program components written with hyperlinks; (c) TSL Editor – the work environment that facilitates text editing, and includes features such as syntax coloring, error highlighting, content assist, hyperlinking, quick fixes, and breakpoints; and (d) Error log - tabular frame aggregating the error markers and warnings detected by the compiler.

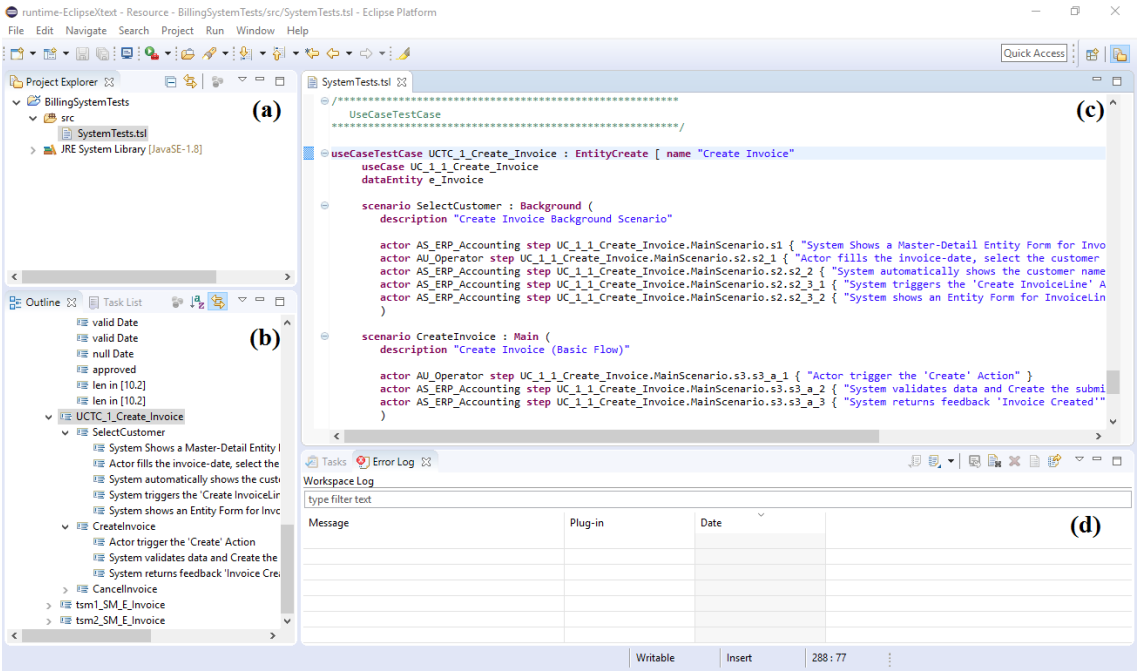


Figure 4.1: Eclipse Editor instance for TSL implementation (a) Project Explorer (b) Outline display (c) TSL editor (d) Errors and Warnings Log

4.2 TSL Overview

TSL, acronym for "Test Specification Language", is a model-based testing approach in which test cases are constructed directly from a RSL model through a DSL. Maintaining the same formal specification, TSL specifies various black-box test cases in a syntactic manner similar to that expressed by RSL. This approach, in the same way as RSL, is implemented through Xtext Eclipse-based platform allowing to take advantage of all the features previously mentioned, regarding the Eclipse IDE Editor.

By applying Black-Box testing design techniques, TSL allows the construction of three different requirement test patterns, from the perspective of functional system tests, that are expressed in the RSL approach. Each one of these constructs were conceived through the analysis of the RSL System view (Chapter 2.2.2), which describes the details concerning the functional aspects of the

system. The RSL system level contain the packages: DataEntity, StateMachine, UseCases and Actors. This lead to the creation of three different test type constructions, more specifically:

- **Domain Analysis -** equivalence partitioning and boundary value analysis for the definition of structural test values, performed over the RSL data entities;
- **Use Case Testing -** derivation of test scenarios from the various process flows expressed by the RSL Use Cases;
- **State Machine Testing -** covering the sequence of state transactions from the RSL event-based State Machines.

Each one of these constructs is dependent on the RSL system-level packages, as expressed on the figure below, Figure 4.2. In the next sections we will present each of these test classes constructed by TSL, including the language style and the terms dependent on the RSL grammar.

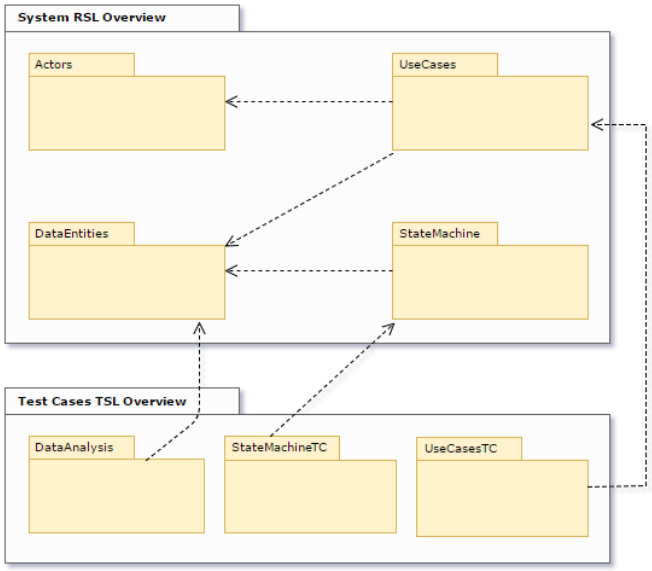


Figure 4.2: TSL package overview and dependencies with RSL System viewpoint

4.2.1 Domain Analysis - Test Cases modeling in TSL

Domain analysis is a classic software test design technique used to identify efficient and effective test values. A domain is the set of all inputs accepted by the SUT. A sub-domain, also known as equivalence class, is a partition of the entity domain defined by boundary conditions. Since the full input domain is typically too large to be used as test inputs, it is partitioned into a finite number of equivalence classes. Each one of these classes represent a set of valid inputs (values expected to return a non error output) or invalid inputs (values that should not be accepted by the SUT). As illustrated on the Figure 4.3.

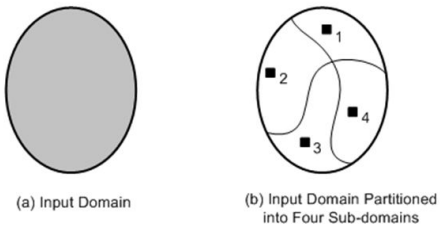


Figure 4.3: a) Full domain of test input.
b) Selection of one input from each equivalence class.

This technique splits in two main black-box approaches, equivalence class and boundary value testing. In Equivalent Class Partitioning the domain is divided trough a set of constraints into several partitions (sub-domains) that can be considered similar. Then, test values from each sub-domain are generated in order to exercise the SUT. Meanwhile, Boundary Testing is the process of testing values within the extremes ends (boundaries between sub-domains of the test values). In this context, a boundary is where two partitions meet.

TSL models domain analysis cases from the RSL System-level view *DataEntities*. Each data entity, also named as "informational entities", is an abstraction and encapsulation from the physical implementation of database tables. They describe the type of data handled by the system and respective constraints. This allows the definition of respective equivalence classes and test values.

These terms inherent to the construction of Domain Analysis Test Cases are summarize in the Table 4.1. This linguistic pattern contains the attributes for its definition (more specifically: <id>, <name>, <type>, <dataEntity>, <attribute>, <paritionClass> and <value>). The Table presents for each one of this attributes its respective name (e.g., id, name, type), multiplicity (e.g., "1" mandatory, "0..*" zero or more, "0..1" optional value), a description and its type value (e.g., ID, String, Boolean, Enumerated Type, Reference key value).

The Enumerated Type (enum) indicates a set of values within predefined constants. The variable must be equal to one of the values that have been predefined for it. In this case, type term indicates the type of domain analysis coverage criteria applied to the test case (which can be *EquivalencePartitioning* and *BoundaryValueAnalysis*, respectively). A Reference key value is done by specifying the Term Attribute inside square brackets. For instance, [DataEntity] indicates a key reference value for the DataEntity linguistic pattern and [Attribute] references an Attribute from the specified Data Entity.

Table 4.1: Overview of Domain Analysis Test Cases terms

Term	Mult	Type/Values	Description
id	1	ID	unique identifier of the element
type	1	DomainAnalysisType: EquivalencePartitioning, BoundaryValueAnalysis	type of the element
name	1	String	name of the element
description	0..1	String	general description of the element
dataEntity	1	[DataEntity]	reference for the data entity in test
attribute	1..* (n)	[Attribute]	reference for the data entity attributes in test
partitionClass	n	String	partition class applied for the test value attribute
value	n	String	input test value for the respective partition class

The Xtext TSL linguistic style for the representation of Domain Analysis Test Cases, containing the terms previously mentioned, is shown on the next Listing:

Listing 4.1: Domain Analysis Test Case linguistic style in TSL

```
DomainAnalysisTestCase:
        'entityTestCase' name=ID ':' type=DomainAnalysisTestCaseType '['
                'name' nameAlias=STRING
                'dataEntity' entity= [DataEntity]
                (testAttributes+=TestAttribute+)
                ('description' description=STRING)?
        ']';

// Domain Analysis Test Case Term Type
enum DomainAnalysisTestCaseType:
        EquivalencePartitioning | BoundaryValueAnalysis;

// Definition of a test attribute
TestAttribute:
        'testAttribute' attribute= [Attribute | QualifiedName] '(' '
            partitionClass'
    classe=STRING 'value'  value=STRING ')';
```

4.2.2 Use Case - Test Cases modeling in TSL

Functional test cases can be derived from Use Cases. Use Cases describe the interactions between an actor and the system through a sequence of steps. In this context, an actor represents a user, or a sub-system, that interacts with the system. Each use case is conceived by the perspective of a user, instead of the system, illustrating the possible actions that can be performed from the user's point of view (not the system's expected inputs/outputs). A use case shall yield the following properties [Zie06]:

- Are initiated by an actor;
- Model an interaction between an actor and the system;
- Describe a sequence of actions;
- Capture functional requirements;
- Provide some observable result or value to an actor;
- Represent a complete and meaningful flow of events.

To better understand how test cases are generated from use cases, it is important to have a brief look to the requirements pyramid, Figure 4.4. This pyramid illustrates a top-down traceability level from product needs to test cases, over the perspective of use cases. On the highest level are the clients or stakeholders needs for the system, from where a group of features are defined. A feature, according to the IEEE 829 standard is "a distinguishing characteristic of a software item (e.g., performance, portability, or functionality)" [IEE08]. Mapping features to use cases and supplementary specifications, or Supp Spec (description of non-functional details), is a one to many relationship. Considering the use cases, the functional requirements of the system, test cases should be developed for each use case scenario. Scenarios are identified by selecting the distinct flows/paths from a use case (e.g., the basic/main flows, alternate flows or exception flows). This way, a use case scenario represents an instance of a use case, a complete flow action from the use case.

The diagram below, Figure 4.4 b), exemplifies a typical flow transaction from a use case. Presenting a Basic/Main flow which describes the normal functioning of the use case (straight bold arrow). Meanwhile, alternate or exception flows can occur leading to a deviation from the main execution (curved arrows). Some of these may return to the normal basic path of events (e.g., Alternate Flow 1 and 3) and others may end the use case (e.g., Alternate Flow 2 and 4).

TSL defines Use Cases Test Cases from the RSL System-level view *Actors* and *UseCases*. Each test contains multiple scenarios, which are derived from the various flows of each RSL Use Case. A scenario encompasses of a group of steps and must be executed by an actor, which are also derived from the RSL System-level view. The terms used to specify a Use Case Test Case are compiled in the following table 4.2. This linguistic pattern contains the attributes for its

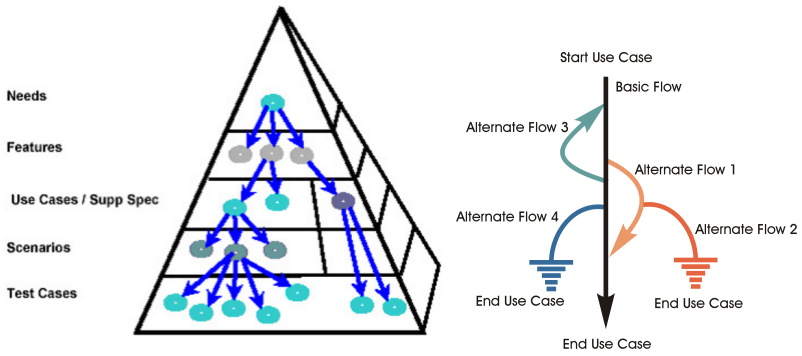


Figure 4.4: a) Requirements traceability pyramid [Zie06]
b) Basic and Alternate flows (Scenarios) for a Use Case [Heu01]

definition (more specifically: <id>, <name>, <type>, <useCase>, <dataEntity>, <background>, <description>, <scenario>, <scenarioType>, <actor>, <step> and <stepDefinition>).

This construct begins by defining the test set, including ID, name and the use case type. Then it encompasses the references keys [UseCase] indicating the Use Case in which the test is proceeding, background [UseCase] in the circumstances of prevailing event flow that take place before the current Use Case, [DataEntity] referring to a possible data entity that is managed throughout the action flow. Considering the flow diagram, for each test case multiple scenarios can be retrieved. For each of this scenarios it is specified a name, the Scenario Type (Main, Alternative or Exception flow, respectively), and the set of steps needed to be performed. For each step it must be indicated the actor who performs it [Actor], a reference to the Use Case step [Step] and a step definition, describing the action executed.

The Table below presents for each one of this attributes its respective name (e.g., id, name, type), multiplicity (e.g., "1" mandatory, "0..*" zero or more, "0..1" optional value), a description and its type/value (e.g., ID, String, Boolean, Enumerated Type, Reference key value).

Table 4.2: Overview of Use Case Test Cases terms

Term	Mult	Type/Values	Description
id	1	ID	unique identifier of the element
name	1	String	name of the element
type	1	TermType	type of the element
useCase	1	[UseCase]	id of the UseCase in Test
description	0..1	String	general description of the element
background	0..1	[UseCase]	reference for the background UseCase
dataEntity	0..1	[DataEntity]	DataEntity manipulated during the transaction
scenario	1..* (n)	String	description of the test scenario
scenarioType	n	scenarioType: Main, Alternative, Exception, Background	type of the test scenario
actor	n..*	[Actor]	Actor who executes the step
step	n..*	[Step]	id of the Step in execution
stepDefinition	0..*	String	description of step

The Xtext TSL linguistic style for the representation of Use Case Test Cases, containing the terms previously mentioned, is shown on the next Listing:

Listing 4.2: Use Case Test Case linguistic style in TSL

```
UseCaseTestCase:
        'useCaseTestCase' name=ID ':' type=UseCaseType '['
                'name' nameAlias=STRING
                'useCase' useCase=[UseCase]
                ('feature' description=STRING)?
                ('background' background=[UseCaseTestCase])?
                ('dataEntity' entity=[DataEntity])?
                scenarios+=TestScenario+
        ']';

// RSL Use Case Term Type definition
enum UseCaseType:
        EntityCreate | EntityRead | EntityUpdate | EntityDelete | EntityReport
    | EntityDashboard | EntityOther | EntitiesManage | EntitiesBrowse |
    EntitiesSearch | EntitiesReport | EntitiesDashboard |
    EntitiesInteropImport |EntitiesInteropExport | EntitiesInteropSync |
    EntitiesInteropSendMessage |EntitiesInteropServiceInvocation |
    EntitiesMapShow | EntitiesMapInteract | EntitiesOther  | Other;

// Test Scenario definition
TestScenario:
        'scenario' name=ID ':' type=ScenarioType '('
```

```
                'description' description=STRING
                testSteps+= TestStep+
                ')';

// Scenario Term Type
enum ScenarioType:
        Main | Alternative | Exception;

// Test Step definition
TestStep:
        'actor' actor+=[Actor] 'step' step+=[Step | QualifiedName]('{'
            stepDefinition=STRING'}')?;
```

4.2.3 State Machine - Test Cases modeling by TSL

A state machine is a model that describes the dynamic system behavior over a given data entity (or object) over its life-cycle. An object is considered an isolated entity that interacts with the system environment, detecting events and responding through actions, resulting in a change of state. A state machine comprises the following four main parts:

- a finite number of states, abstract situation which the software may occupy during the data entity life cycle;
- transactions between two states, that is a valid sequence from one state to another;
- events, a particular input or stimulus to the system, that causes a transaction;
- actions, the system's output or operation result from a transaction.

Generating test cases from state machines models can be advantageous, since they describe the complete behavior and respective data flow from an entity during its entire software life cycle. State transition testing is a black-box testing technique used to derive test cases from a finite state machine model representation. State Machine Test cases are defined to exercise the whole system, given a coverage criteria. For instance, test every single state machine state (All State Coverage), cover all valid transition shown in the model (Switch-0), cover all of the pairs of two valid transitions or sets of three transitions (Switch-1 and Switch-2 respectively) or just test a normal object flow, describing a Typical Transaction.

State Machine Test Cases are specified trough TSL by modeling the RSL System-Level view *StateMachine*. This RSL view details all the main aspects that covers state machines: the states, transactions, events and actions. A TSL state machine test case is specified through a sequence of states that needed to be executed. The table below, Table 4.3, display the terms to produce TSL State Machine Test Cases from RSL state machines. This linguistic pattern contains the attributes for its definition, more specifically: <id>, <name>, <type>, <dataEntity>, <stateMachine>, <test-State> and <description>.

This construct begins by defining the test set, including ID, name and the Test Case type (according to the coverage criteria selected) and description. Also contains the references keys [DataEntity], referring to the object that is managed throughout the action flow, and [StateMachine], the respective RSL state machine from which the test cases are specified. The sequence of states to be executed are defined through a reference [State] for the given state of the state machine,

Table 4.3: Overview of State Machine Test Cases terms

Term	Mult	Type/Values	Description
id	1	ID	unique identifier of the element
type	1	TermType	type of the element
name	1	String	name of the element
dataEntity	0..1	[DataEntity]	reference for the data entity
stateMachine	1	[StateMachine]	reference for the state machine
testState	1..*	[State]	reference for the current test state
description	0..1	String	general description of the element

The Xtext TSL linguistic style for the representation of state Machine Test Cases, containing the terms previously mentioned, is shown on the next Listing:

Listing 4.3: State Machine Test Case linguistic style in TSL

```
StateMachineTestCase:
        'stateMachineTestCase' name=ID ':' type=StateMachineTestCaseType '['
                'name' nameAlias=STRING
                'dataEntity' entity= [DataEntity]
                'stateMachine' statemachine= [StateMachine]
                testStates+= TestState+
                ('description' description=STRING)?
        ']';

// State Machine Test Case Term Type
enum StateMachineTestCaseType:
        TypicalTransaction | AllStateCoverge | SwitchCoverage0 |
            SwitchCoverage1;

// Test States sequence definition
TestState:
        'testState' refState+=[State | QualifiedName](',' refState+=[Step |
            QualifiedName])*;
```

4.3 TSL State Machine Automation Testing Tool

Despite TSL allowing for a rigorous and powerful support for specification of test cases based on RSL format, thanks to the Eclipse IDE integration, it can't fully provide automation of test artifacts. The projection of state transactions sequences from the RSL representation of a finite-state machine (FSM) is a laborious task, given that there is no graphical illustration and the user has to trace valid sequences manually. As one of the main objectives from this research, to present an higher degree of automation to the TSL solution, it was elaborated a support tool for visualization and extraction of state machines from a RSL format (Xtext and Excel).

As mentioned before, the sequence of states are defined to exercise the SUT following a certain criteria. In this case it was explored a Switch-0 approach, covering sequences with the maximum valid transactions present in the model. This was achieved through the usage of a Depth-First Algorithm (DFS). It traverses a graph in a depth-ward motion and uses a stack to store vertexes to effectuate the search. When a dead end is ecountered, then it means that a full state transaction (test case) has been disclose.

Given that the presented state machine only produces a unique computation (transaction) for each accepted input strings of symbols, we have a deterministic finite state machine (DFSM). A DFSM is represented by a quintuple $(\Sigma, S, s_0, \delta, F)$, consisting of:

- Σ is the input alphabet, a finite and non-empty set of symbols.
- S is a finite number of non-empty set of states.
- s_0 is an initial state, an element of S.
- δ is the state-transition function $\delta : S \times \Sigma \rightarrow S$
- F is the set of final states, a subset of S.

The pseudocode comprehending the implemented depth-first search algorithm is presented below, Algorithm 1.

Algorithm 1 Depth-First Algorithm

```
1:  procedure DFSWITHSTACK
2:      St ← EmptyStack
3:      V ← s0                                          ▷ Insert initialState
4:      for all edge in S(E) do
5:          visited[edge] ← false
6:      popStack ← false
7:      while St is not Empty do                        ▷ While stack is not empty
8:          N ← findNeihbourByEdge(V)
9:          if N! = null then                           ▷ If next edge is not null
10:             popStack ← false
11:             St.add(N)
12:             V ← N
13:         else if N == null then                      ▷ If next edge is null
14:             if popStack == false then               ▷ If is found a dead-end
15:                 createTestCase(St)
16:             popStack ← true
17:             V ← S.pop()
18:             visited[V] ← true
19:     return testList                                 ▷ ArrayList with the testCases founded
```

Figure 4.5 presents the primary graphic interface of this tool. This panel accommodates the three main options: Read RSL Xtext model, Read RSL Excel model and Run Switch-0 Algorithm, respectively. The body of the panel comprises a text area in order to print out the obtained test cases.

Figure 4.5: State Machine TSL Test Generator - primary graphic interface

When the RSL model is read by the tool it automatically presents the graphic output of state machine (Figure 4.6, right frame). This is handled by the use of JUNG (Java Universal Network/-Graph) Framework [Tea17a], a software library that provides a common and extensible language for modeling, analysis, and visualization of data that can be represented as a graph or network. After loading the model and executing the algorithm, the generated test cases are presented in TSL format at the text area encompassed in the main interface (Figure 4.6, left frame).

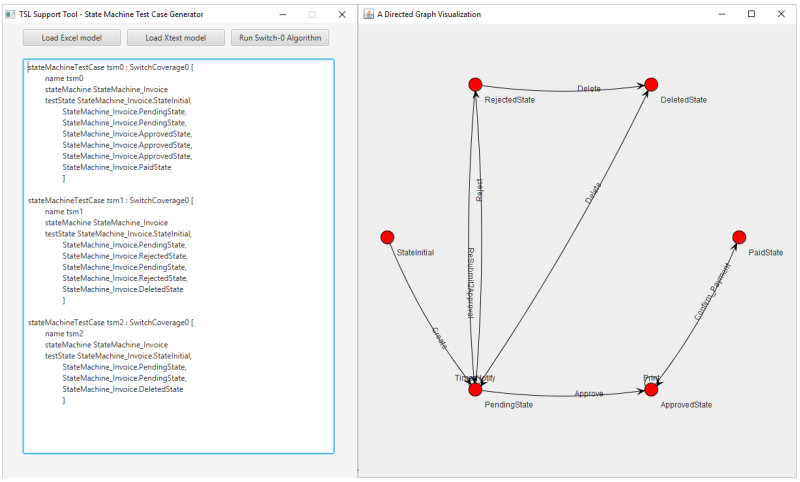


Figure 4.6: State Machine TSL Test Generator - results output and graphic visualization

4.4 RSL Excel Template: TSL extension

RSL, aside from the Eclipse XText-based tool, is also currently supported by other software tools including an Excel spreadsheet template. It allow RSL approach to have a broader usage, since users can adopt a general purpose tool as Microsoft Excel. Therefore it was conceived the “RSL Excel Template” [SSC15], a SRS template based that applies the RSL requirement language approach. The term "IL" stands for Intermediate Language, since the requirements specified don’t hold the same formality and rigor intrinsic to the RSL Xtext grammar. Each one of the views expressed in the RSL Business and System-level is arranged in a different Excel Sheet with a common tabular format.

This template includes two main files, (1) *"RSLIL-ExcelTemplate"* a configurable and customizable template to be used in a project basis; (2) *"RSLIL-Example-BillingSystem"* a simple example based on the specification of the "Billing System", showing a real case use of the first template.

In order to extend the TSL approach to other RSL formats, along with the XText framework, were created three new views in the form of Excel sheets to the “RSL Excel Template”. Each one of this views projects the definition of each one of the previously mentioned test cases types (Domain Analysis, Use Case Testing and State Machine Testing), containing the same linguistic pattern definition. The full definition of each one of these views is presented in the Appendix A.

4.5 Conclusion

This chapter presented the conceived solution to apply the Model-based testing approach into the RSL language. TSL, short-term for "Testing Specification Language", is a language to formally specify test cases based on the SRS model RSL. System tests patterns, and strategies to construct them over the functional requirements, were described allowing a systematization of the system's testing process.

TSL supports both of Xtext and Excel RSL formats. The Xtext based model is handled with the integration of the Eclipse IDE. This work environment implements DSL editor for test construction, covering most important features concerning a language IDE. The Excel RSL format is extended with the creation of three Excel sheets, arranged in a tabular way, for each of this test classes. This grants a broader usage, since testers with no IT background can specify tests using a popular tool as MS-Excel. On the other hand, it loses part of the rigor and formality inherent to the Xtext format.

Additionally it was created a TSL State Machine Support Tool, supplying graphic visualization and extraction of test cases from RSL state machine specifications. This way it was provided an higher degree of automation to the TSL solution.

Chapter 5

Study Case: Billing System

The previous chapter presented the methodology and the solution developed to extract and specify system tests from a RSL model to a new testing domain specific language named TSL. Now it is introduced a case study demonstrating a concrete usage of TSL to specify the different test classes directly from a RSL specification. It is described an information system named "Billing System" and specified test cases examples for each of the test class types designed - Domain Analysis, Use Case Testing and State Machine Testing.

5.1 Billing System Overview

Throughout the RSL research development, in order to illustrate a real case scenario implementation of this requirement language, it was conceived an application example named "Billing System" [SSC15; Rod17]. This fictitious application expresses a very simple management system for customers, products and invoices. It allows the users to handle the tracking of invoices, documents of billable products or services delivered by a seller to a customer. A user is a person with a registered account, being able to access and interact with the system through a specified user profile (user role). The Billing System comprises the user profiles User-customer, User-operator and User-manager.

The User-customer (customer) has the function of handling the customer related data (e.g., create and update customer information or delete its account). Meanwhile, a User-operator (employee) is responsible for managing customers, products and invoices. This actor, apart from executing the same actions as the customer user, is also in charge of controlling Products (e.g., insert/delete new instances in the database, update its information and manage product's stock) and Invoices. It is his responsibility to approve/reject invoices before they are issued and sent to respective customers, update invoices and invoice lines, print or export them in other formats, etc. Lastly, the User-manager (manager) is in charge of monitoring the general configurations of

the system. It comprises the management of the enterprise information, definition of value-added taxes (VAT) and consult the sales reports over a product or a customer.

The next diagram (Figure 5.1) by Alberto, Daniel and Tiago (2015) explicits a Use Case diagram for this system. It contains the respective actors (Customer, Employee and Manager) and their interactions (actions) with the system, defined into four main categories: Manage Customers, Manage Products, Manage Invoices and General Configurations.

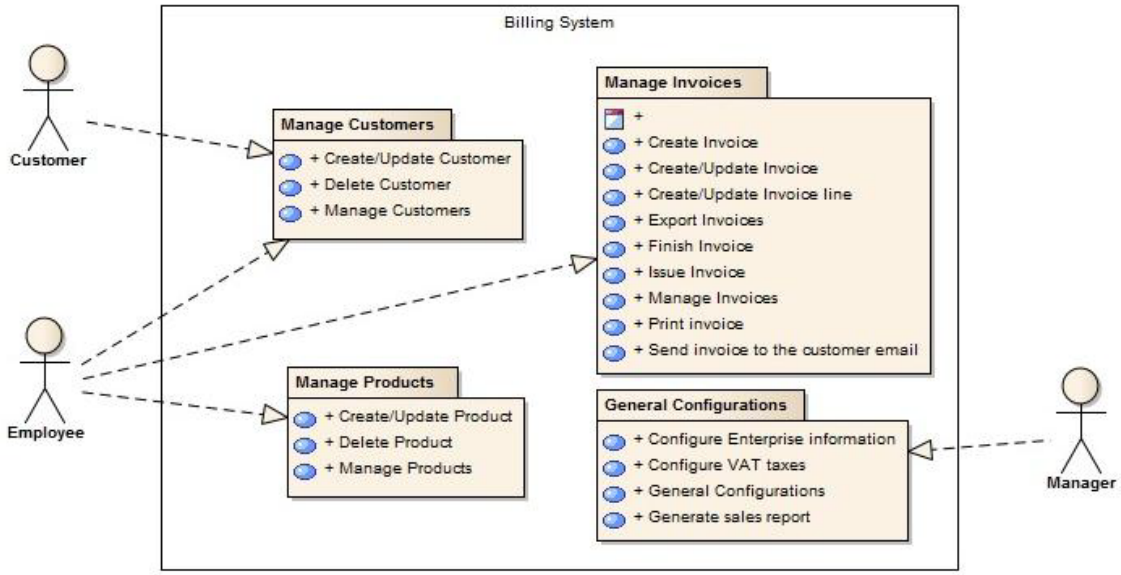


Figure 5.1: Billing System Use Case model [SSC15]

In order to simplify and to not overextend, the TSL test cases presented are mainly focused on the *Create Invoice* Use Case. So it is presented a Domain Analysis example over the Invoice data entity (e_Invoice), the definition of test case scenarios for this particular Use Case and extraction of state sequences over the invoice data entity life-cycle State Machine.

5.1.1 Domain Analysis Cases

In the Billing System context, an invoice is a commercial document related to a sale transaction between a seller to a buyer (customer). For each invoice the system shall indicate the products, quantities, agreed prices for products or services the seller had provided the buyer. The RSL linguistic construct defines an Invoice entity (e_Invoice) with its attributes, more specifically: Integer <ID>, Integer foreign key <customerID>, date value <dateCreation>, date value <dateApproaval>, date value <datePaid>, date value <dateDeleted>, boolean <isApproved>, decimal totalValueWithoutVAT> and the decimal <totalValueWithVAT>.

The Listing 5.1 presents a domain analysis case for a valid instance of an Invoice. Partition classes for each e_Invoice attribute were generated by employing Equivalence Partitioning technique, in which test values were taken from valid partition classes. Additionally, other domain test cases could be elaborated to exercise the system with Boundary Analysis test values or invalid

instances of this data entity. For example, a Invoice object with invalid Customer ID, price values out of range (including extremely long and negative values), date of payment inferior to the creation date, etc.

Listing 5.1: State Machine Test Case linguistic style in TSL

```
entityTestCase DA_e_Invoice : EquivalencePartitioning [ name "Valid Invoice"
        dataEntity e_Invoice
                testAttribute e_Invoice.ID (
                        partitionClass "valid ID" value '1')
                testAttribute e_Invoice.customerID (
                        partitionClass "valid ID" value '1')
                testAttribute e_Invoice.dateCreation (
                        partitionClass "valid Date" value "1/06/2017")
                testAttribute e_Invoice.datePaid (
                        partitionClass "valid Date" value "30/06/2017")
                testAttribute e_Invoice.dateDeleted (
                        partitionClass "null Date" value "")
                testAttribute e_Invoice.isApproved (
                        partitionClass "approved" value "True")
                testAttribute e_Invoice.totalValueWithoutVAT (
                        partitionClass "len in [10.2]" value '100,00')
                testAttribute e_Invoice.totalValueWithVAT (
                        partitionClass "len in [10.2]" value '80,00')]
```

5.1.2 Use Case Test Cases

The creation of invoices, as expressed by the Use Case diagram, is a task by the responsibility of the user-operator. The employee shall fill the master data form with respective information, including invoice-date and respective customer, then the system shall automatically show the customer data in the info area. At this point the user can choose between two options, validate the data and submit the generated Invoice or cancel the action and abort the Use Case.

The next Listing 5.2 illustrates the creation of a TSL Use Case test based on the RSL Use Case specification for Create Invoice. It starts with a common set of steps, designated through a Background scenario, and then it splits in two different scenarios. The first one consists on the main flow event, when a new Invoice is created, the second one represents an alternative path where the operation is canceled. This exemplifies a complete interaction between the user AU_Operator (Employee) and the System. Also, since the TSL test contains a reference key for a data entity (in this case e_Invoice) it allows the extension of Domain Analysis test technique. This way a tester can execute the multiple scenarios specified by TSL and resort to the test data created for the given data entity.

Listing 5.2: State Machine Test Case linguistic style in TSL

```
useCaseTestCase UCTC_1_Create_Invoice : EntityCreate [ name "Create Invoice"
        useCase UC_1_1_Create_Invoice
        dataEntity e_Invoice

        scenario SelectCustomer : Background (
               description "Create Invoice Background Scenario"

               actor AS_ERP_Accounting step UC_1_1_Create_Invoice.MainScenario
                   .s1 { "System Shows a Master-Detail Entity Form for Invoice
                   /InvoiceLines" }
               actor AU_Operator step UC_1_1_Create_Invoice.MainScenario.s2.
                   s2_1 { "Actor fills the invoice-date, select the customer
                   from a selection-list" }
               actor AS_ERP_Accounting step UC_1_1_Create_Invoice.MainScenario
                   .s2.s2_2 { "System automatically shows the customer name,
                   NIF and address in a Customer info area" }
               actor AS_ERP_Accounting step UC_1_1_Create_Invoice.MainScenario
                   .s2.s2_3_1 { "System triggers the 'Create InvoiceLine'
                   Action" }
               actor AS_ERP_Accounting step UC_1_1_Create_Invoice.MainScenario
                   .s2.s2_3_2 { "System shows an Entity Form for InvoiceLines,
                    with the available Actions (Create, Cancel)" }
               )

        scenario CreateInvoice : Main (
               description "Create Invoice (Basic Flow)"

               actor AU_Operator step UC_1_1_Create_Invoice.MainScenario.s3.
                   s3_a_1 { "Actor trigger the 'Create' Action" }
               actor AS_ERP_Accounting step UC_1_1_Create_Invoice.MainScenario
                   .s3.s3_a_2 { "System validates data and Create the
                   submitted Invoice/InvoiceLines" }
               actor AS_ERP_Accounting step UC_1_1_Create_Invoice.MainScenario
                   .s3.s3_a_3 { "System returns feedback 'Invoice Created'" }
               )

       scenario CancelInvoice : Alternative (
               description "Cancel Invoice (Alternate Flow)"

               actor AU_Operator step UC_1_1_Create_Invoice.MainScenario.s3.
                   s3_b_1 { "Actor trigger the 'Cancel' Action" }
               actor AS_ERP_Accounting step UC_1_1_Create_Invoice.MainScenario
                   .s3.s3_b_2 { "System aborts operation" }
               )
       ]
```

5.1.3 State Machine Test Cases

The life-cycle over the invoice data entity (e_Invoice) is represented in RSL under the format of a state machine. As already expressed (Chapter 4.2.3), a state machine allows to model the behavior of a data entity (object) as a set of event-driven transactions from a state to another when triggered by an event.

In this case, afterwards an invoice (the result of a sale between a seller to a client of a product/service) is created enters a pending state. It is an employee's responsibility to approve, delete or reject an issued invoice, guiding to the respective invoice's state of Approved, Rejected and Deleted. Afterwards, an approved invoice can be printed or exported to other formats (e.g., electronic email) before the user-operator formally confirms the customer's payment, reaching the final Paid State. Also, a rejected invoice can either be resubmitted, returned to the Pending State, or wiped out from the system (Deleted State).

The next Figure 5.2 illustrates the graphic output resultant from the read of the RSL invoice state machine by the TSL support tool, Chapter 4.3. It comprises its states (State StateInitial, PendingState, ApprovedState, RejectedState, PaidState and DeletedState) and its respective transactions.

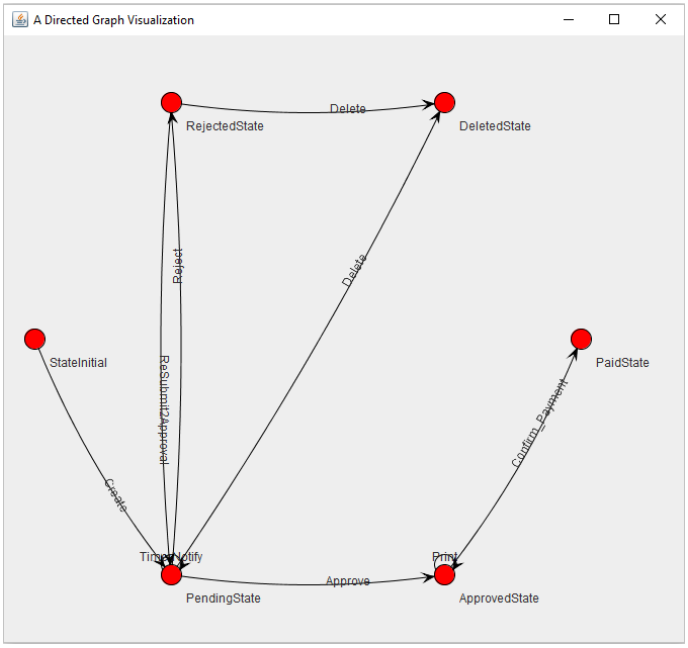


Figure 5.2: Invoice State Machine graph.

The Listing 5.3 illustrates a concrete representation of a TSL State Machine test case. Following the use of the Support Tool algorithm, it was obtained three test sequences covering the Switch-0 Coverage. The first test case comprehends a regular invoice routine from the Initial state to a Paid state, crossing the Pending and Approved states in between. The second sequence of state transactions designates an Invoice flow from the Initial state to Delete state, but also going through the Rejected state. For last, the third case defines an invoice deleted right after its creation.

Listing 5.3: State Machine Test Case linguistic style in TSL

```
stateMachineTestCase tsm1_SM_E_Invoice : SwitchCoverage0 [
        name "tsm1_SM_E_Invoice"
        stateMachine StateMachine_Invoice
        testState StateMachine_Invoice.StateInitial,
                StateMachine_Invoice.PendingState,
                StateMachine_Invoice.PendingState,
                StateMachine_Invoice.ApprovedState,
                StateMachine_Invoice.ApprovedState,
                StateMachine_Invoice.ApprovedState,
                StateMachine_Invoice.PaidState]

stateMachineTestCase tsm2_SM_E_Invoice : SwitchCoverage0 [
        name "tsm2_SM_E_Invoice"
        stateMachine StateMachine_Invoice
        testState StateMachine_Invoice.StateInitial,
                StateMachine_Invoice.PendingState,
                StateMachine_Invoice.RejectedState,
                StateMachine_Invoice.PendingState,
                StateMachine_Invoice.RejectedState,
                StateMachine_Invoice.DeletedState]

stateMachineTestCase tsm3_SM_E_Invoice : SwitchCoverage0 [
        name "tsm3_SM_E_Invoice"
        stateMachine StateMachine_Invoice
        testState StateMachine_Invoice.StateInitial,
                StateMachine_Invoice.PendingState,
                StateMachine_Invoice.PendingState,
                StateMachine_Invoice.DeletedState]
```

Chapter 6

Conclusion and Future Work

Model-Based Testing is a software testing technique in which test cases are generated from a model of the SUT. This way it is possible to obtain test cases from requirements specifications to achieve automation and systematization of the testing process.

RSL, short-term for "Requirements Specification Language", is a formal language to support and improve the production of system requirements specification (SRS). With the use of controlled natural language it guides the production of understandable and coherent textual sentences. Closing the gap of requirements representation and natural language, which is the root of many requirements quality problems.

This dissertation describes the TSL, short-term for Test Specification Language, a Model-Based Testing approach to specify test cases, through the perspective of system tests, from a RSL software model. Functional test cases are mapped from the various RSL package-system views, containing several constructs that describe the system behavior, concretely: Actor view, DataEntity view, UseCase view and StateMachine view. This lead to the creation of three main test constructs by applying of black-box test design techniques. More specifically:

- **Domain Analysis -** equivalence class partitioning and boundary-value analysis for the definition of structural test values from the various Data Entities of the system;
- **Use Case Test Cases -** derivation of test scenarios from the various process flows expressed by the Use Cases;
- **State Machine Test Cases -** covering the sequence of state transactions from event-based State Machines.

TSL systematize the test developing process of both Xtext and Excel RSL formats. Xtext-based format is handled with the integration of the powerful Eclipse IDE. This work environment implements DSL editor for test construction, covering most important features concerning IDE, granting TSL a semi-automated way to formally build test cases. Comprehending syntax-aware

editor, immediate feedback, incremental syntax checking, suggested corrections, auto-completion, and so on, it provides great assist for composing tests. Excel RSL format is extended with the creation of three Excel sheets, arranged in a tabular way, for each of this test classes. This grants a broader usage, since testers with no IT background can specify tests using a general tool as MS-Excel. On the other hand, it loses part of the rigor and formality inherent to the Xtext format.

Model-Based Testing is conventionally associated with automated creation of test cases. In this research was also explored a full generation of test cases based on RSL State Machine specifications. That way it was created a TSL State Machine Support Tool, allowing a graphic visualization and obtainment of state sequences based on the Switch-0 coverage criteria. This allowed a bigger degree of automation to the final solution, which was also one of the main objectives of this research.

It is important to address, although TSL is a powerful and rigorous MBT framework to build tests, it does not solve all testing problems. Comprised in one of the seven testing principles, "testing is context dependent" [IG11] denotes that the test methodology is not a matter of tooling, being employed differently within situation and condition. Conventional test design techniques are supported by TSL, granting higher level of systematization, but cannot be fully replaced. The tester must still have a complete knowledge of the system's domain to perform effective and efficient tests.

Other constraint inherent to MBT techniques in general (consequently to TSL) since tests are based on software models, this may also hold defects manually introduced by its creator. If the model is not properly verified and validated, these errors may propagate to generated test outputs. RSL prevents most of these issues by documenting requirements in a textual controlled language, but it is still important to verify an validate the model before defining tests. Furthermore, considering modern Agile software development techniques for fast delivery and continuous improvement (e.g., Behavior-Driven Development, Test-Driven Development, Acceptance Test–driven Development), models are not static artifacts, they change and evolve over time. Modifications done in a RSL model cause alterations over the TSL file as well. This requires constant attention to updates effectuated over the RSL model.

The study case "Billing System", a fictitious invoice management application, allowed to illustrate how the several test case constructs can be represented in a concrete and practical scenario. Demonstrating that, as executable requirements specifications, functional tests can be easy to "read, write, execute, debug, validate, and maintain" [KK14].

6.1 Future Work

This dissertation was the first initiative towards Model-based testing in the RSL language. It provided a starting point for future research with a succinct and methodical overview regarding requirements patterns and their construction, from a functional system test perspective, encountered

in this specific model. For further development work it would be relevant to automate more processes for test case generation and creation of automatic test execution with integration of external test frameworks.

The developed TSL State Machine Support Tool generates test cases based on a Switch-0 coverage, it would also be interesting to implement algorithms based on other coverage criteria (e.g., Switch-1 or Switch-2). Aside from that, one could explore the possibility of more automated processes, for instance: the generation of domain analysis test data by combinatorial generation of values for each attribute (e.g., constrains on possible attribute values) and extraction of test scenarios based on the varies flows expressed by Use Cases.

The generated tests specified in TSL can be executed manually by a tester to exercise the SUT and discover possible errors in the system. It would be interesting for further research to explore the integration of TSL files, of real developed systems, with test frameworks to provide automatic execution of those tests. For example, exploration of tools such Cucumber [Cuc17] or Specflow [Tea17b] which enables the automatic execution of tests in a plain-text language called Gherkin. Cucumber is a popular tool employed in various languages including Java, JavaScript, and Python. Meanwhile, Speclow is an open source solution for .NET projects. This way it would be possible to provide an oracle for the tests, determining whether they passed or failed.

References

[AD97] Larry Apfelbaum and John Doyle. Model Based Testing. *Procedings of the 10th Internation Software Quality Week*:1–14, 1997. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.1342%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[BF14] Pierre Bourque and Richard E. Fairley. *Guide to the Software Engineering - Body of Knowledge*. 2014, page 346. ISBN: 0-7695-2330-7. DOI: 10.1234/12345678. arXiv: arXiv:1210.1833v2. URL: www.swebok.org.

[BR] Lorenzo Bettini and Reshma Raman. *Implementing Domain-Specific Languages with Xtext and Xtend Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices Implementing Domain-Specific Languages with Xtext and Xtend*. ISBN: 978-1-78646-496-5. URL: www.packtpub.com.

[Bur02] Ilene Burnstein. *Practical Software Testing*. 2002, pages 1–732. ISBN: 0387951318. DOI: 10.1007/b97392. arXiv: arXiv:1011.1669v3. URL: http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract.

[Buu15] Robin A. ten Buuren. Domain-Specific Language Testing Framework. (October), 2015.

[Cuc17] The company behind Cucumber & Cucumber Pro. Cucumber: simple, human collaboration. 2017. URL: https://cucumber.io/ (visited on 06/20/2017).

[DC12] Sandeep Dalal and Rajender Singh Chhillar. Case Studies of Most Common and Severe Types of Software System Failure. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8):341–347, 2012. URL: http://www.ijarcsse.com/docs/papers/8%7B%5C_%7DAugust2012/Volume%7B%5C_%7D2%7B%5C_%7Dissue%7B%5C_%7D8/V2I800209.pdf.

[DD12] David De Almeida Ferreira and Alberto Rodrigues Da Silva. RSLingo: An information extraction approach toward formal requirements specifications. *2012 2nd IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012 - Proceedings*:39–48, 2012. DOI: 10.1109/MoDRE.2012.6360073.

[DD13a] David De Almeida Ferreira and Alberto Rodrigues Da Silva. RSL-IL: An interlingua for formally documenting requirements. *2013 3rd International Workshop on Model-Driven Requirements Engineering, MoDRE 2013 - Proceedings*:40–49, 2013. DOI: 10.1109/MoDRE.2013.6597262.

[DD13b] David De Almeida Ferreira and Alberto Rodrigues Da Silva. RSL-PL: A linguistic pattern language for documenting software requirements. *2013 3rd International Workshop on Requirements Patterns, RePa 2013 - Proceedings*:17–24, 2013. DOI: 10.1109/RePa.2013.6602667.

[DKV00] a. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(June):26–36, 2000. ISSN: 03621340. DOI: 10.1145/352029.352035.

[Fag01] Michael E. Fagan. *Advances in software inspections*. In *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*. Manfred Broy and Ernst Denert, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pages 335–360. ISBN: 978-3-642-48354-7. DOI: 10.1007/978-3-642-48354-7_14. URL: http://dx.doi.org/10.1007/978-3-642-48354-7_14.

[FD12] David De Almeida Ferreira and Alberto Rodrigues Da Silva. Formally specifying requirements with RSL-IL. *Proceedings - 2012 8th International Conference on the Quality of Information and Communications Technology, QUATIC 2012*:217–220, 2012. DOI: 10.1109/QUATIC.2012.30.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN: 0321712943, 9780321712943.

[FR] David De Almeida Ferreira and Alberto Rodrigues. Obtaining Formal Requirements Representations with the RSLingo Approach.

[Heu01] Jim Heumann. Generating Test Cases From Use Cases. *The Rational Edge*, 6(01), 2001. URL: http://goose.ycp.edu/%7B%5C%%7D7B%7B~%7D%7B%5C%%7D7Dweddins/papers/usecasepapers/paper1.htm%7B%5C%%7D5Cnhttp://goose.ycp.edu/%7B~%7Dweddins/papers/usecasepapers/paper1.htm.

[Ibe13] Marcel Ibe. Decomposition of test cases in model-based testing. *CEUR Workshop Proceedings*, 1071:40–47, 2013. ISSN: 16130073.

[IBM06] IBM. Why do Project Fail ? Why is requirements management so critical ? (August 2006), 2006.

[IEE08] IEEE. *Std. 829-2008: Standard for Software and System Test Documentation*, volume 2008 of number July. 2008, page 132. ISBN: 9780738157467. URL: https://goo.gl/UBf7JN.

[IEE90] IEEE. IEEE_SoftwareEngGlossary.pdf, 1990. URL: www.mit.jyu.fi/ope/kurssit/.../IEEE%7B%5C_%7DSoftwareEngGlossary.pdf.

[IG11] ISTQB and GTB. Certified Tester Foundation Level Syllabus:85, 2011.

[Int12] International Software Testing Qualifications Board. Certified Tester - Advanced Level Syllabus Test Analyst:82, 2012.

[ISO00] ISO 9126. Information technology — Software product quality. *Iso/Iec Fdis 9126-1*, 2000:1–26, 2000. ISSN: 10774866. DOI: 10.1002/(SICI)1099-1670(199603)2:1<35::AID-SPIP29>3.0.CO;2-3. arXiv: arXiv:1011.1669v3. URL: http://www.cse.unsw.edu.au/%7B~%7Dcs3710/PMmaterials/Resources/9126-1%20Standard.pdf.

[Iso94] N Z S Iso. Australian / New Zealand Standard Quality management and quality assurance — Vocabulary, 1994.

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

[JP93] Matthias Jarke and Klaus Pohl. Establishing visions in context: towards a model of requirements processes, 1993.

[KBB+17] Anne Kramer, Robert V Binder, Robert V Binder, Anne Kramer, Bruno Legeard, and All Rights. Model-based Testing User Survey : Results, 2017.

[KK14] Tariq King and Tariq King. Functional Testing with Domain- Specific Languages, 2014.

[Kul05] Victor V. Kuliamin. Multi-paradigm Models as Source for Automated Test Construction. *Electronic Notes in Theoretical Computer Science*, 111(SPEC. ISS.):137–160, 2005. ISSN: 15710661. DOI: 10.1016/j.entcs.2004.12.002.

[Mat08] Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008. ISBN: 8131716600, 9788131716601.

[McC09] Steve McConnell. *Software Project Survival Guide*, volume 2009. 2009, page 306. ISBN: 0735637385. URL: http://books.google.com/books?id=NvQRDTbBZOEC%7B%5C&%7Dpgis=1.

[Mic16] Zoltán Micskei. Model-based testing (mbt). 2016. URL: http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html (visited on 05/20/2017).

[MPN+17] Rodrigo M. L. M. Moreira, Ana Cristina Paiva, Miguel Nabuco, and Atif Memon. Pattern-based gui testing: bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*, 27(3):e1629–n/a, 2017. ISSN: 1099-1689. DOI: 10.1002/stvr.1629. URL: http://dx.doi.org/10.1002/stvr.1629. e1629 stvr.1629.

[Pat01] Ron Patton. *Software Testing*. 2001, page 389. ISBN: 0672319837. URL: http://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:No+Title%7B%5C#%7D0.

[Poh10] Klaus Pohl. The Requirements Engineering Framework. *Requirements Engineering: Fundamentals, Principles, and Techniques-Solution-Oriented Requirements*:41–58, 2010.

[Rod14] Alberto Rodrigues. Quality of Requirements Specifications - A Framework for Automatic Validation of Requirements. *Proceedings of the 16th International Conference on Enterprise Information Systems*:96–107, 2014. DOI: 10.5220/0004951900960107. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0004951900960107.

[Rod17] Alberto Rodrigues Da Silva. Linguistic Patterns and Styles for Requirements Specification: The RSL/Business-Level Language, 2017.

[RSU+14] Authors Alex Randell, Eric Spellman, William Ulrich, Jeff Wallk, Reviewers Mike Clark, Eric Elliott, and Whynde Melaragno. Leveraging Business Architecture to Improve Business Requirements Analysis. (March), 2014.

[Spa15] John Spacey. What is a requirement? 2015. URL: http://simplicable.com/new/requirement (visited on 05/20/2017).

[SSC15] Alberto Rodrigues Silva, Daniel Serrão, and Tiago Catariano. RSL-IL Excel Template : A System Requirement Specification based on the RSL-IL Language , V1 . 1. (October), 2015. URL: https://github.com/RSLingo/RSL-IL-ExcelTemplate/blob/master/RSLIL-ExcelTemplate-TechnicalReport-v1.1.pdf.

[Sve17] Miro Spoenemann Sven Efftinge. Xtext - language engineering made easy! 2017. URL: https://eclipse.org/Xtext/ (visited on 04/28/2017).

REFERENCES

[TBS11] M Timmer, H Brinksma, and M I A Stoelinga. Model-Based Testing. *Software and Systems Safety: Specification and Verification*, 30:1–32, 2011. ISSN: 1874-6268.

[TC13] David R. Tobergte and Shirley Curtis. *Domain-SpecificLanguages*, volume 53 of number 9. 2013, pages 1689–1699. ISBN: 9788578110796. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3.

[Tea17a] The JUNG Framework Development Team. Jung - java universal network/graph framework. 2017. URL: http://jung.sourceforge.net/ (visited on 05/04/2017).

[Tea17b] The Specflow Development Team. Specflow - binding business requirements to .net code. 2017. URL: http://specflow.org/ (visited on 06/20/2017).

[ULP+06] Mark Utting, Bruno Legeard, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-Based Testing. *Software Testing, Verification and Reliability*, 22(April):297–312, 2006. ISSN: 1099-1689. DOI: 10.1002/stvr.456. URL: http://onlinelibrary.wiley.com/doi/10.1002/stvr.456/abstract%7B%5C%%7D5Cnhttp://onlinelibrary.wiley.com/store/10.1002/stvr.456/asset/stvr456.pdf?v=1%7B%5C&%7Dt=hiy99iid%7B%5C&%7Ds=dd768678914b374853367307dd34

[Wie03] Karl Eugene Wiegers. *Software Requirements*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2003. ISBN: 0735618798, 9780735618794.

[Zie06] P. Zielczynski. Traceability from use cases to test cases. *The Rational Edge*, 2006:1–22, 2006. DOI: 10.1109/CIT.2007.116. URL: http://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:Traceability+from+Use+Cases+to+Test+Cases%7B%5C#%7D0%7B%5C%%7D5Cnhttp://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:Traceability+from+use+cases+to+test+cases%7B%5C#%7D0.

Appendix A

RSL Excel Template - TSL Views

In this appendix is presented a complete overview of the Test Specification Language (TSL) views conceived to support the RSL Excel Template. Each one of the constructed Excel sheets contain the definition of a functional system test class, respectively: Domain-Analysis, Use Case Testing and State Machine Testing.

Back to Index

<SystemS> Tests Specification

Test Cases from Domain Analysis (for System S1)

System
Entities
Test Attributes

S_SystemS

Id (*)	**Name (*)**	**Entity Ref (*)**	**Description**	**Type (*)**		
				Test Attributes		
				Attribute Ref	**Partition Class**	**Test Value**
test1_VAT	test1_VAT	E_VAT	Test Null VAT Category	BoundaryValueAnalysis		
				ID	null	null
				name	null	null
			Test Attributes	value	null	null
test2_VAT	test2_VAT	E_VAT	Test Valid VAT Category	EquivalencePartitioning		
				ID	valid	e_Vat
				name	valid	CategoryA
			Test Attributes	value	valid	10
test1_Product	test1_Product	E_Product	Product	BoundaryValueAnalysis		
				ID	null	null
				name	null	null
				price	null	null
				VATCategory	null	null
			Test Attributes	value	null	null
test2_Product	test2_Product	E_Product	Product	EquivalencePartitioning		
				ID	valid	1
				name	valid	ProductX
				price	valid	100
				VATCategory	valid	e_Vat
			Test Attributes	value	valid	90

Figure A.1: Excel view for specification of TSL domain analysis tests (TSL.domainAnalysis sheet).

Back to Index

<SystemS> Tests Specification

Test Cases Definition from Use Cases (for System S1)

System
Use Case
Scenarios
Steps

S_SystemS

TC_TestCase1

TEST CASE DEFINITION					SCENARIOS			
					Scnario Type	Scenario Description		
Id (*)	Name (*)	Type (*)	Background TC	Description	Scenario Name	Step Id	Subject	Step Description
Main Scenario	Main Scenario	FR_X	-	Description of the test case				
					MainScenario	Scanario1 description		
					Scenario1	1	Actor	PreConditions of the TestCase
						2a	Actor	...
						2b	Actor	...
						...	...	...
						3	Actor	PostConditions of the TestCase
					AlternativeScenario	Scenario2 Description		
					Scenario2	1	Actor	PreConditions of the TestCase
						2a	Actor	...
						2a1	Actor	...
						...	...	...
						3	Actor	PostConditions of the TestCase

TC_1_Manage_Invoices

					Type	Description		
Id (*)	Name (*)	Type (*)	Background TC	Description	Name	Step Id	Subject	Step Description
TC_1	Manage Invoices	EntitiesManage	-	Manage Invoices, including create/manage Invoice-lines, and optionally create customer records				
					AlternativeScenario	Cancel Manage Invoices		
					Cancel	1	System	Shows a 'List of Entity Form' for Invoices, with the available Actions
						2a	Actor	Triggers the 'Close' Action
						3	System	Terminates use case
					MainScenario	Search Manage Invoices		
					Search	1	System	Shows a 'List of Entity Form' for Invoices, with the available Actions
						2_b_1	Actor	fills the 'Search' textbox with a search criteria, defined as a string expression
						2_b_2	Actor	trigger the 'Search' Action
						2_b_3	System	shows a refreshed list of invoices based on the search criteria
					AlternativeScenario	Filter Manage Invoices		
					Filter	1	Actor	Shows a 'List of Entity Form' for Invoices, with the available Actions
						2_b_1	Actor	trigger the 'Filter' Action
						2_b_2	Actor	fills the 'Filter' fiels with a search criteria, defined as a logical expression...
						2_b_3	System	trigger the 'Filter' Action
						2_b_4	System	shows a refreshed list of invoices based on the Filter criteria

TC_2_Create_Invoices

					Type	Description		
Id (*)	Name (*)	Type (*)	Background TC	Description	Name	Step Id	Subject	Step Description
TC_3	Create Invoices	FR_2	TC_2	Create Invoice, including create/manage Invoice-lines, and optionally create customer				
					MainScenario	Select Invoice		
					Select Customer	1	System	Shows a Master-Detail Entity Form for Invoice/InvoiceLines, with the available Actions
						2	Actor	Fills the Entity Form, with the following sub-steps:
						2_1	Actor	Fills the invoice-date, select the customer from a selection-list
						2_2	System	Automatically shows the customer name, NIF and address in a Customer info area
						2_3	Actor	Creates multiple InvoiceLines as need, based on the following sub-steps:
						2_3_1	Actor	Triggers the "Create InvoiceLine' Action
						2_3_2	System	Shows an Entity Form for InvoiceLines, with the available Actions (Create, Cancel)
					MainScenario	Create Invoice Scenario		
					Create	3_a_1	Actor	Triggers the 'Create' Action
						3_a_2	System	Validates data and Create the submitted Invoice/InvoiceLines
						3_a_1	System	Returns feedback 'Invoice Created'
					AlternativeScenario	Cancel Invoice Scenario		
					Cancel	3_b_1	Actor	Triggers the 'Cancel' Action
						3_b_2	System	Cancels operation

Figure A.2: Excel view for specification of TSL use case tests (TSL.usecases sheet).

Back to Index

<SystemS> Tests Specification

Test Cases Definition from State Machines (for System S1)

System
State Machine
Test Cases

S_SystemS

SM_State Machine

Test Case Definition				Test Cases			
				TC1	**TC2**	**TC3**	**TC4**
Id (*)	**Name (*)**	**Description**	**Entity (*)**	**TypicalTransaction**	**AllStateCoverage**	**SwitchCoverage0**	**SwitchCoverage1**
Main Scenario	Main Scenario	Description of the test case	-	Initial State	Initial State	Initial State	Initial State
				Next State	Next State	Next State	Next State
				...	...	...	...
				Final State	Final State	Final State	Final State

StateMachine_Invoice

				TC1	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
Id (*)	**Name (*)**	**Description**	**Entity (*)**	**TypicalTransaction**	**AllStateCoverage**	**SwitchCoverage0**	**SwitchCoverage0**	**SwitchCoverage1**	**SwitchCoverage1**
SM_E1	**StateMachine_Invoice**	**StateMachine of Entity In**	**E_Invoice**	StateInitial	StateInitial	StateInitial	StateInitial	StateInitial	StateInitial
				PendingState	PendingState	PendingState	PendingState	PendingState	PendingState
				DeletedState	RejectedState	PendingState	RejectedState	RejectedState	PendingState
					PendingState	ApprovedState	DeletedState	PendingState	DeletedState
					ApprovedState	ApprovedState			

Figure A.3: Excel view for specification of TSL state machine tests (TSL.statemachines sheet).