A Test Specification Language for Information Systems based on Data Entities, Use Cases and State Machines

Alberto Rodrigues da Silva[1], Ana C. R. Paiva[2,3], and Valter E. R. da Silva[2]

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa,
Lisboa, PORTUGAL
alberto.silva@tecnico.ulisboa.pt
[2] Faculdade de Engenharia da Universidade do Porto,
[3] INESC TEC
Rua Dr. Roberto Frias, s/n 4200-465 Porto PORTUGAL
apaiva@fe.up.pt, svalter.ribeiro@gmail.com

Abstract. Testing is one of the most important activities to ensure the quality of a software system. This paper proposes and discusses the TSL (Test Specification Language) that adopts a model-based testing approach for both human-readable and computer-executable specifications of test cases. TSL is strongly inspired on the grammar, nomenclature and writing style as defined by the RSLingo RSL, which is a rigorous requirements specification language. Both RSL and TSL are controlled natural languages that share common concepts such as data entities, use cases and state machines. However, by applying black-box functional testing design techniques, TSL includes and supports four complementary testing strategies, namely: domain analysis testing; use case tests; state machine testing; and acceptance criteria. This paper focuses on the first three testing strategies of TSL. Finally, a simple but effective case study illustrates the overall approach and supports the discussion.

Keywords: Test Specification Language (TSL), Test Case Specification, Model based Testing (MBT), Test Case Generation.

1 Introduction

One of the most important activities to increase the quality of software systems is software testing. It is known that up to 50 percent of the total development costs are related to testing [3] and that about 30 to 60 percent of the total effort within a project is spent on testing [7].

Model-based testing (MBT) is one technique that systematizes and automates more of the testing process [11, 18, 27, 35–38, 42]. Test cases are generated from a model of a given Software System. These models or specifications vary in nature: they can be more or less abstract and represented textually [12] and/or graphically [10]; they can describe the functionalities and other qualities or goals [16] of the system. However, often these models do not exist, which demand they

have to be developed from scratch; or, if they exist, they are described in a very informal way, which does not allow to derivate automatically test cases from it. Also, write system and acceptance tests manually is ineffective since tests are hard to write and costly to maintain. Hence, the existence of requirements specification, defined with controlled natural languages, may enable the derivation of test cases directly from such specifications. So, leveraging domain specific languages (DSLs) for functional testing can provide several benefits. For example, Robin Buuren recognizes in his work "Domain-Specific Language Testing Framework" three major quality aspects [2]: (i) Effectiveness, because it reduces the time of test development, since tests can be generated from a model; (ii) Usability, because it is easier to produce such test specification, considering the support provided by the work environment; and (iii) Correctness, because it makes system tests clearer by giving testers programmatic and strictly defined rules, leading to fewer errors.

This research proposes and discusses the TSL (Test Specification Language) that adopts a model-based testing approach for both human-readable and computer-executable specification of test cases. TSL is strongly inspired on the grammar, nomenclature and writing style as defined by the RSLingo RSL, which is a rigorous requirements specification language [24, 25]. TSL applies black-box functional testing design techniques and includes and supports four complementary testing strategies, namely, (i) domain analysis testing: a testing strategy that uses techniques such as equivalence partitioning and boundary value analysis for the definition of structural data values; (ii) use case tests: a testing strategy that defines tests based on the scenarios defined for each use case; (iii) state machine testing: a testing strategy that traverses the state machine expressed in RSL according to different coverage criteria, e.g., cover all states; and (iv) acceptance criteria: a more general testing strategy based on the BDD (Behavior Driven Development) approach and very popular with user stories. This paper focus on the first three testing strategies.

To illustrate the overall TSL approach we use a fictitious information system (the "BillingSystem") that is partially described as a variety of informal requirements. That description is to some extent deliberately incomplete, vague and inconsistent as it is common in real-world situations. From that description, RSL requirements are defined and afterwards some TSL tests are derived and/or manually defined. The TSL definition of test cases for data entities, use cases and state machines and the corresponding generated Gherkin specifications are presented.

This paper updates and extends the one presented in [32] that introduced the general idea and architecture of the TSL language. This paper restructures that former paper describing first the overall approach in a more general way, and then further presenting and discussing the language based on an illustrative example. Furthermore this paper describes in more detail the constructs Data Entities, Use Cases and State Machines, and originally discusses the respective test strategies, namely the domain analysis testing (based on the construct *DataEntityTestCase*); use case testing (based on the construct *UseCaseTestCase*); and

state machine testing (based on the construct *StateMachineTestCase*). Finally, this paper also adds the description of the TSL for specifying tests based on use cases and it extends the illustrative case study with TSL test cases based on use cases, and so, showing the strong relation between these three constructs at both requirement and testing levels.

This paper is organized in 6 sections. Section 2 introduces and overviews the RSL language, by introducing its bi-dimensional multi-view architecture, based on abstraction levels and concerns. Section 3 gives a very short introduction to the concepts of Cucumber and Gherkin. Section 4 presents and discusses the TSL constructs and views, namely tests based on data entities use cases and state machines. Section 5 presents a case study to illustrate the overall approach. Finally, Section 6 presents the conclusion and identifies issues for future work.

2 RSL Overview

It is known that natural language, although the most used notation in the requirements documents, it is prone to produce ambiguities and inconsistencies that are difficult to validate or transform automatically. RSLingo is a long-term research initiative that proposed an approach to use simplified natural language processing techniques as well as human-driven techniques for capturing relevant information from ad-hoc natural language requirements specifications and then applying lightweight parsing techniques to extract domain knowledge encoded within them [4]. This was achieved through the use of two original languages: the RSL-PL (Pattern Language) [5], designed for encoding RE-specific linguistic patterns, and RSL-IL (Intermediate Language), a domain specific language designed to address RE concerns [6]. Based on these two languages and the mapping between them, it is possible to extract, analyze and convert the initial written knowledge in natural language into a more structured format, reducing its original ambiguity and creating a more rigorous SRS document.

Based on the former languages [5, 6, 14, 15, 17, 18, 21, 29, 30], Silva et al. designed, more recently, a broader and more consistent language, called "RSLingo's RSL" (or just "RSL" for the sake of brevity). RSL is a control natural language to help the production of SRSs in a systematic, rigorous and consistent way [24, 25]. RSL is a process- and tool-independent language, i.e., it can be used and adapted by different users and organizations with different processes/methodologies and supported by multiple types of software tools.

RSL provides several constructs that are logically arranged into views according to two viewpoints: the abstraction level (Levels) and the specific RE concerns (Concerns) they address. These views are organized according to two abstraction levels: business and system levels; and to five concerns: context, active structure, behavior, passive structure and requirements (Fig. 1).

At the business level, RSL supports the specification of the following business-related concerns: (1) the people and organizations that can influence or will be affected by the system; (2) business processes, events, and flows that might help to describe the business behavior; (3) the common terms used in that business

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

Fig. 1. RSL Levels and Concerns [24, 25, 32].

domain; and (4) the general business goals of stakeholders regarding the value that the business as well the system will bring. Considering these concerns, RSL business level comprises respectively the following views: *Stakeholders* (active structure concern), *BusinessProcesses* (behavior concern), *Glossary* (passive structure concern), and *BusinessGoals* (requirements concern). In addition, the references to the systems used by the business, as well as their relationships, can also be defined at this level (context concern).

On the other hand, at the system level, RSL supports the specification of multiple RE specific concerns, namely by the adoption of the following constructs: (1) to describe the actors that interact with the system; (2) to describe the behavior of some systems data entities, namely based on state machines; (3) to describe the structure of the system, namely based on data entities and data entity views; and (4) several to specify the requirements of the system according different styles. Considering these concerns, the system level respectively comprises the following views: *Actors* (active structure concern); *StateMachines* (behaviour concern); *DataEntities* and *DataEntityViews* (passive structure concern); and multiple types of Requirements such as *SystemGoals*, *QualityRequirements* (QRs), *Constraints*, *FunctionalRequirements* (FRs), *UseCases*, and *UserStories* (requirements concern). In addition, all these elements and views should be defined in the context of a defined *System* (context concern).

The following subsections give some detail regarding the RSL elements that are then used (in section 4) to support the TSL tests specifications.

2.1 Data Entities

A *DataEntity* in RSL (see Spec. 1) denotes an individual structural entity that might include the specification of attributes, foreign keys and other (check) data constraints. A *DataEntity* is classified by a type and an optional subtype.

An Attribute denotes a particular structural property of the respective *DataEntity*. An attribute has an *id*, *name*, *type* (e.g., Integer, Double, String, Date) and

optionally the specification of its *multiplicity*, *default value* (i.e., the value assigned by default in its creation time), *values* (i.e., a list of possible values, e.g., enumeration values separately by semicolons), and *is not null* and *is unique* constraints.

```
'dataEntity' name=ID ':' type=DataEntityType (':' subType=DataEntitySubType)?'['
        ('name' nameAlias=STRING)
        ('isA' super=[DataEntity])?
        (attributes+=Attribute)+
        (primaryKey=PrimaryKey)?
        (foreignKeys+=ForeignKey)*
        (checks+=Check)*
        ('description' description=STRING)?
']';

'attribute' name=ID ':' type=AttributeType ('(' size = DoubleOrInt ')')? '['
       ('name' nameAlias=STRING)
       ('multiplicity' multiplicity=Multiplicity)?
       ('defaultValue' defaultValue=STRING)?
       ('values' values=STRING)?
       (notNull='NotNull')?
       (unique='Unique')?
       ('description' description=STRING)?
']';
```

Spec. 1. Definition of Data Entities in RSL.

2.2 Use Cases

UseCases view in RSL defines the uses cases of a system under study (see Spec. 2). Traditionally a use case means a sequence of actions that one or more actors perform in a system to obtain a particular result [33]. However, the RSLs *UseCase* construct extends such general and vague definition considering some additional aspects, namely [25]:

- A use case shall be classified by a set of use case types;
- A use case can be applicable to a cluster of data entities, called *DataEntityView*;
- A use case shall define at least the actor that initiates it and, optionally, other actors that might participate; these actors can be end-users, external systems or even timers;
- A use case can define pre-conditions and post-conditions;
- A use case can define several actions that may occur in its context;
- A use case can define "includes" relations to other use cases;
- A use case can define several extensions points available in its context;
- A use case can extend the behavior of other use case (the target use case) in its specific extension point;
- The behavior of a use case can be detailed by a set of scenarios that are also classified as main, alternative or exception scenario; by definition a use case

can only have one main scenario and zero or more alternative and exception scenarios;
- A use case scenario is defined by a set of sequential or parallel steps;
- A use case step is classified by a set of types and defined as simple or complex step;

```
'useCase' name=ID ':' type=UseCaseType '['
  ('name' nameAlias=STRING)
  ('actorInitiates' actorInitiates=[Actor])
  ('actorParticipates' actorParticipates+=RefActor)?
  ('dataEntityView' dEntityView=[DataEntityView])?

  ('precondition' precondition=STRING)?
  ('postcondition' postcondition=STRING)?

  (actions= UCActions)?
  (extensionPoints= UCExtensionPoints)?

  (includes= UCIncludes)?
  (extends+= UCExtends)*

  ('stakeholder' stakeholder=[Stakeholder])?
  ('priority' priority=PriorityType)?
  ('description' description=STRING)?

  scenarios+=Scenario*
']';

'scenario' name=ID ':' type=ScenarioType '['
  ('name' nameAlias=STRING)
  ('executionMode' mode=('Sequential'|'Parallel'))?
  ('description' description=STRING)?
  (steps+=Step*)
']';

'step' name=ID ':' type=StepOperationType (':' subType=StepOperationSubType )?'['
  (simpleStep= SimpleStep | subSteps+= Step+ | ifSteps+= IfStep* )
']';
```

Spec. 2. Definition of Use Cases in RSL.

2.3 State Machines

The *StateMachines* view in RSL defines the behavior of *DataEntities* in their relationships with use cases (see Spec. 3). A *StateMachine* is necessarily assigned to just one *DataEntity* and classified as simple or complex depending on the number of states and transitions involved (e.g., a *StateMachine* with more than 4 states might be classified as Complex). A *StateMachine* includes several states corresponding to the situations that a *DataEntity* may be find itself during its life cycle (e.g., as shown in Spec. 11 states like *Pending*, *Approved*, *Rejected*). In addition, a state can be defined as initial (*isInitial*) or final (*isFinal*). Several actions can be defined when a *DataEntity* enters (*onEntry*) or exits (*onExit*) the respective state. Moreover, several use cases actions (*actions*) can occur on the

DataEntity when it is in a given state, and the occurrence of these actions can optionally imply a state transition (*nextState*).

```
'stateMachine' name=ID ':' type=StateMachineType '['
  'name' nameAlias=STRING
  'dataEntity' entity= [DataEntity]
  ('description' description=STRING)?
  states= States
']';


State:
'state' name=ID
  (isInitial ?= 'isInitial')?
  (isFinal ?= 'isFinal')?
  ('onEntry' onEntry= STRING)?
  ('onExit' onExit= STRING)?
  (':' (actions+= RefUCAction))? (actions += RefUCAction)* ;

RefUCAction:
  ('useCaseAction' action= [UCAction] ('nextState' nextstate= [State])?);
```

Spec. 3. Definition of State Machines in RSL.

3 Gherkin and Cucumber Overview

Behavior-Driven Development (BDD) is a software development methodology in which a software application is specified and designed describing how its behavior should appear to an external observer [26]. In BDD, acceptance tests are first written describing the behavior of the system from a user's point of view, and then these acceptance tests are reviewed and approved so that code development can begin.

Acceptance tests written in a behavior-driven style can be executed by the testing tool Cucumber [39]. Cucumber reads the acceptance tests and validates that the software does what is expected. Because tests are automatically executed by Cucumber, it is easy to maintain the specifications up-to-date.

Gherkin [40] is a simple language that Cucumber is able to understand. Gherkin allows to define tests in an easily readable format. Gherkin tests are organized into features. Each feature is made up of a collection of scenarios defined by a sequence of steps and following a Given-When-Then (GWT) rule. A simple example is illustrated in Spec. 4, more information can be obtained, for example, in [40].

4 TSL Approach and Language

The original goal of this research is to develop an approach to support the specification and generation (whenever relevant) of software tests defined in TSL,

```
Feature: Login Action
Scenario: Successful Login with Valid Credentials
        Given User is on Home Page
         When User Navigate to LogIn Page
          And User enters UserName
          And Password
         Then Message displayed Login Successfully
```

Spec. 4. Simple Test Case Example in Gherkin.

directly from requirements specifications originally defined in RSL. It is intended to achieve the following goals: (i) extend the RSLingo approach with the support for testing activities; (ii) define a set of strategies that would allow generating test cases from the RSL constructs; and (iii) automate the test case generation process.

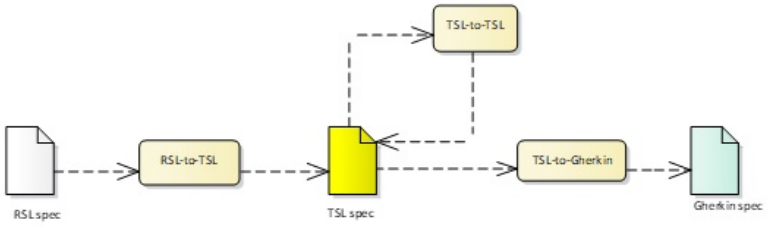

Fig. 2. TSL Based Approach [32].

Fig. 2 suggests the proposed approach. First, RSL requirements specifications are the input for the RSL-to-TSL transformation that generates TSL specifications. Second, based on predefined strategies, these TSL specs can be expanded and generated into other TSL specs (e.g., for increasing the system testing domain with more test cases). Third, the TSL specs are the input for the TSL-to-Gherkin transformation that generates Gherkin specifications, and ultimately these specs can be used for documentation purposes or even for testing execution.

As illustrated in Fig. 3, a TSL specification is a combination of two different types of elements. First, the *TestSupportSpecs* package includes *TestSupportSpec* elements such as *DataEntities*, or *StateMachines*. These elements are a simplified version of the equivalent elements supported by the RSL language (e.g., the TSL *DataEntity* element is a simplified version of the RSL *DataEntity*). These TSL *TestSupportSpec* elements can be authored manually but usually shall be generated from the RSL specs.

Second, the *TestSuite* package includes *TestCase* elements such as *DataEntityTestCase*, *UseCaseTestCase* or *StateMachineTestCase*. Each *TestCase* shall be defined as *Valid* or *Invalid* and shall have a dependency to a respective *Test-*

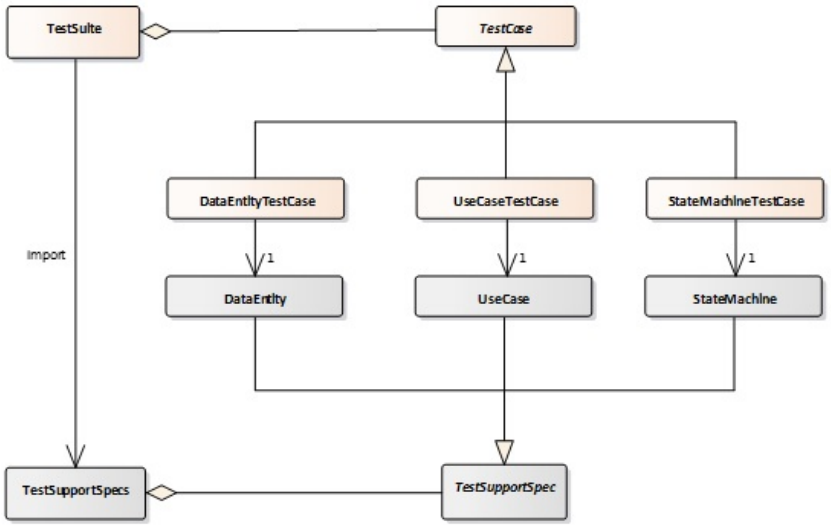

Fig. 3. Metamodel of the TSL general architecture (partial view) [32].

SupportSpec, e.g., a *StateMachineTestCase* has a dependency to the respective *StateMachine*. These *TestCase* elements can be generated by the RSL-to-TSL and TSL-to-TSL transformations, but usually shall be also authored and refined by the software testers.

TSL allows specifying various black-box test cases in a syntactic manner similar to that expressed by RSL. In addition, TSL allows to systematize the test developing process with both Xtext-based and Excel RSL formats. Xtext based format is handled with the integration of the Eclipse IDE [1]. This environment provides an editor for test construction, covering most important features concerning IDE, granting TSL a semi-automated way to formally specify test cases. This Eclipse-based tool provides great assist for composing tests, namely comprehends a syntax-aware editor with features like immediate feedback, incremental syntax checking, suggested corrections, and auto-completion. On the other hand, the RSL/TSL Excel template [34] is extended with the creation of some Excel sheets, arranged in a tabular way, for each of the provided test types. 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 formal grammar defined in Xtext.

TSL supports the specification of the following tests:

– *DataEntityTestCases* can be defined by applying equivalence class partitioning and boundary value analysis [31] over RSL *DataEntities*;

– *UseCaseTestCase* can be defined by exploring multiple sequences of steps defined in RSL use cases' scenarios, and also by associating data values to the involved data entities;

– *StateMachineTestCases* can be defined by applying different algorithms to traverse the state machine defined in RSL, so that it shall be possible to build different test cases that correspond to different paths through the state machine.

4.1 Data Entity Test Cases

Data Entity test cases are based on classic input domain analysis test design techniques known as "equivalence class partitioning" and "boundary value analysis" [31].

Since most of the times it is unfeasible to test all possible values of an input domain, equivalent class partitioning divides the input domain into classes (or data entities in the RSL/TSL terminology) assuming that the behavior of the software is the same for any value inside a class. Hence, the test cases are designed so as to test one randomly chosen value of each class.

Knowing that the probability of finding failures is higher for boundary values, boundary value analysis chooses the boundaries of the classes for test input data instead of randomly chosen values.

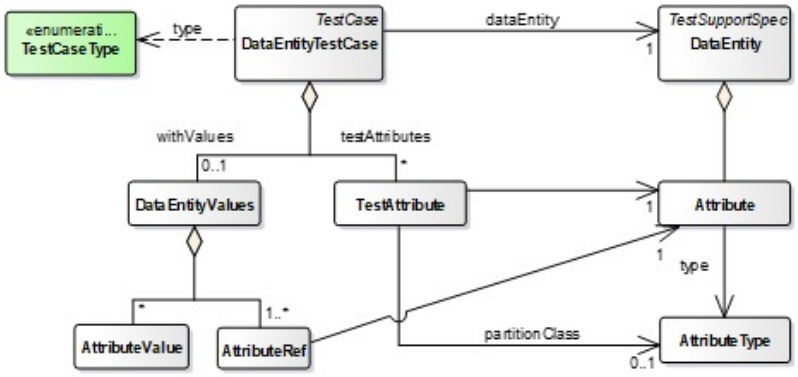

Fig. 4. Metamodel of the TSL DataEntityTestCase definition (partial view) [32].

As shown in Fig. 4, a *DataEntity* keeps information about a specific data entity and its attributes; for each attribute it keeps information about its type, size, among others. Based on this information, it is possible to define equivalence classes and test input data. For example, consider an entity with an attribute A of type real and with one decimal place within [2.3, 25.4], according to equivalence class partitioning, the tester should test valid and invalid input values. So, for this particular case, the tester could define a valid input inside the interval, e.g., 15.2, and invalid inputs outside the interval, e.g., 1.2 and 30.3.

The benefit of the TSL is that it builds a view with all the entities and attributes for which the tester should define test input data. In case of sequential

attribute values (such has numbers), it is also possible to apply boundary value analysis to define test input data. For instance, considering the same attribute A between [2.3, 25.4] the tester should define test input data for the boundaries: 2.2, 2.3, 2.4 (for one boundary) and 25.3, 25.4, 25.5 (for the other boundary).

As illustrated in Fig. 4 , a *DataEntityTestCase* refers to just one *DataEntity* and defines a combination of values that are associated to its respective attributes. These values can be defined individually at an attribute basis (using the *TestAttribute* object) or as a table of values associated to multiple attributes (using the *Values* object). Each *DataEntityTestCase* shall be defined as *Valid* or *Invalid* type depending on the validity of such values.

4.2 Use Case Test Cases

A use case is a description of a particular use of the system by an actor (a user of the system). It helps defining 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.

Use Case tests (Fig. 5) are derived from the various process flows expressed by a RSL Use Case. TSL defines Use Cases Tests 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.

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. For each test case multiple scenarios can be defined. For each of these scenarios it is specified a *name*, the scenario *Type* (Main, Alternative or Exception flow), and the *set of steps* needed to be performed. For each step, it shall be indicated the actor who performs it [Actor], or alternatively the system responsible for it and a step definition, informally describing the action step executed.

4.3 State Machine Test Cases

A state machine is a model that describes the behavior of a system as a whole or more commonly of a given data entity (or object) throughout its life-cycle. A RSL state machine allows to represent the behavior of a data entity as a set of event-driven actions from a state to another when triggered by a given use case action. In addition, from a state machine defined in RSL, it is possible to apply different algorithms that traverse the state machine according to different test coverage criteria, such as, all states or all transitions or others [41]

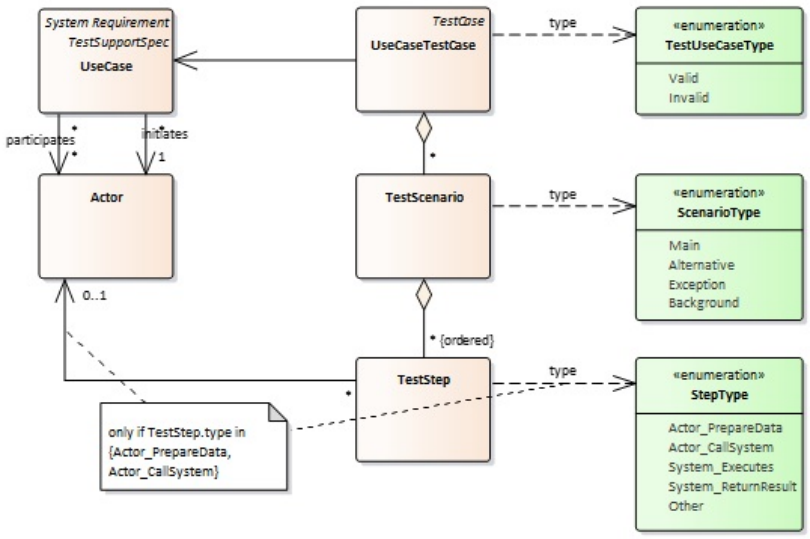

Fig. 5. Metamodel of the TSL UseCaseTests definition (partial view).

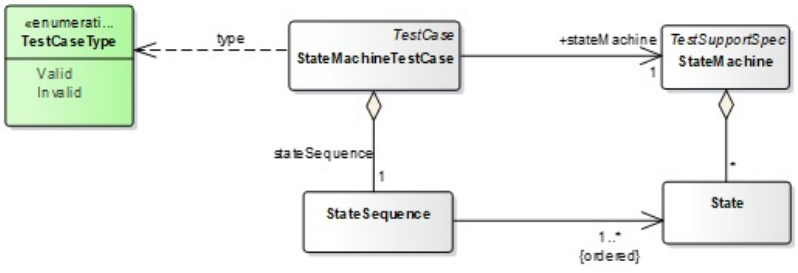

Fig. 6. Metamodel of the TSL StateMachineTestCase definition (partial view) [32].

As illustrated in Fig. 6, a *StateMachineTestCase* specifies the State Machine to which is applied and an ordered sequence of states to traverse (i.e., a *StateSequence*). Finally, this *StateMachine TestCase* shall be defined as Valid or Invalid type depending if that sequence of states are semantically valid or not.

5 Case Study: Billing System

To illustrate the use of the TSL language, we introduce an informal description of the fictions Billing System.

Informal Requirements of the Billing System

BillingSystem is a system that allows users to manage customers, products and invoices. A user of the system is someone that has a user account and is assigned to one or more user roles, such as user, user-operator, user-manager and user-administrator [...].

User-operator is responsible for managing customers and invoices. System shall allow user-operator to create/update information related to customers and invoices [...].

The creation of invoices is a shared task performed by the user-operator and the user-manager. System shall allow user-operator to create new invoices (with respective invoice details). Before sending an invoice to a customer, the invoice shall be formally approved by a user-manager. Only after such approval, the user-operator shall issue and send that invoice electronically by e-mail and by regular post. In addition, for each invoice, the user-operator needs to keep track if it is paid or not [...].

User-manager shall be responsible for approving invoices before they are issued and sent to their customers. User-manager shall allow monitoring the process of creating, approving and payments invoices. User-manager shall approve or reject invoices [...].

5.1 TSL: Data Entity Tests

An invoice is a commercial document related to a sale transaction between a seller and a buyer (customer). For each invoice the system shall indicate the products, quantities, agreed prices for products or services the seller had provided to the buyer. Each product has a price with and without the respective VAT. The VAT (value-added tax) is a type of general consumption tax that is collected incrementally, based on the surplus value, added to the price on the work or the product at each stage of production. Specification in Spec. 5 shows a TSL specification of some of these entities, namely the e_VAT, e_Product and e_Customer data entities.

Based on this data entities' specification it is possible to define and also to generate some data entity test cases. Spec. 6 shows some of these tests defined for the *e_VAT* data entity. First, *detVAT1* is defined as a valid test case

```
DataEntity e_VAT "VAT Category" : Reference [
  attribute VATCode "VAT Code" : Regex [isNotNull isUnique]
  attribute VATName "VAT Class Name" : String(30) [isNotNull]
  attribute VATValue "VAT Class Value" : Decimal(2.1) [isNotNull]
  primaryKey (VATCode)
  description "VAT Categories"]

DataEntity e_Product "Product" : Master [
  attribute ID "Product ID" : Integer [isNotNull isUnique]
  attribute Name "Name" : String(50) [multiplicity "1..2" description "Product Name"]
  attribute valueWithoutVAT "Price Without VAT" : Decimal(16.2) [isNotNull ]
  attribute valueWithVAT "Price With VAT" : Decimal(16.2) [isNotNull ]
  attribute VATCode "VAT Code" : Integer [isNotNull]
  attribute VATValue "VAT Value" : Decimal(2.2) [isNotNull]
  primaryKey (ID)
  foreignKey e_VAT(VATCode)
  description "Products"]

DataEntity e_Customer "Customer" : Master [
  isEncrypted
  attribute ID "Customer ID" : Integer [isNotNull isUnique]
  attribute Name "Name" : String(50) [isNotNull isUnique]
  attribute fiscalID "Fiscal ID" : String(12) [isNotNull isUnique]
  attribute BankID "Bank ID" : Regex
  attribute phone "Phone #" : String(12) [isNotNull isUnique]
  attribute image "Image" : Image
  primaryKey (ID)
  check ck_Customer1 "ValidFiscalID(fiscalID)"
  description "Customers"]
```

Spec. 5. Example of a TSL (partial) specification of data entities.

```
DataEntityTestCase detVAT1 : Valid [
    dataEntity e_VAT
    testAttribute  VATCode (partitionClass Integer values "0;1;2;3" isUnique)
    testAttribute VATValue (partitionClass Decimal(2.2)  values "0; 0,06; 0,13; 0,23" )
    message "Correct VAT values (1)" ]

DataEntityTestCase detVAT2 "detVAT2" : Valid [
    dataEntity e_VAT  withValues (
    |   VATCode         |  VATValue         |   VATValue    +|
    |   0               |   "No-VAT"        |   0           +|
    |   1               |   "Reduced"       |   0.06        +|
    |   2               |   "Intermediate"  |   0.13        +|
    |   3               |   "Normal"        |   0.23        +| )
    testAttribute  VATCode (partitionClass Integer values "0, 1")
    testAttribute  VATName (partitionClass String)
    testAttribute VATValue (partitionClass Decimal(2.2))
    message "Correct VAT values (2)" ]

DataEntityTestCase  detVAT3 "detVAT3" : Invalid [
    dataEntity e_VAT
    testAttribute VATValue (values "0,08; 0,25" message "Incorrect VAT values")
    testAttribute VATValue (partitionClass Integer message "Incorrect VATValue PartitionClass") ]
```

Spec. 6. Example of a TSL (partial) specification of data entity tests.

and defines two *testAttributes*, which both define a partition class check, valid values, and for the *e_VAT.VATCode* attribute a uniqueness constraint. Second, *detVAT2* is defined as a valid test case but shows a set of relevant attributes with valid values in a table format; this representation is usually the most practical and convenient approach to define such values. In addition, *detVAT2* also defines three *testAttributes*. Third, *detVAT3* is defined as an invalid test case and involves the definition of two *testAttributes*, both with problems referred by their respective messages (i.e., "Incorrect VAT values" and "Incorrect *VATValue* PartitionClass").

Spec. 7 shows the equivalent data entity test case in the Gherkin language.

```
Feature: Management of VAT data entity

Scenario: Valid VAT (1)
Given dataEntity VAT
When e_VAT.VATCode partitionClass is Integer
 And e_VAT.VATCode values are "0;1;2;3;"
 And e_VAT.VATValue partitionClass is Decimal(2.2)
 And e_VAT.VATValue values are "0; 0,06; 0,13; 0,23"
Then Output Correct VAT values (1)

Scenario: Valid VAT (2)
Given dataEntity VAT with values:
   |   e_VAT.VATCode   |   e_VAT.VATName   |   e_VAT.VATValue  |
   |   0               |   "No-VAT"        |   0               |
   |   1               |   "Reduced"       |   0.06            |
   |   2               |   "Intermediate"  |   0.13            |
   |   3               |   "Normal"        |   0.23            |
When e_VAT.VATCode partitionClass is Integer
 And e_VAT.VATName partitionClass is String
 And e_VAT.VATValue partitionClass is Decimal(2.2)
 And e_VAT.VATName values are <e_VAT.VATName>
 And e_VAT.VATValue values are <e_VAT.VATValue>
Then Output Correct VAT values (2)


Scenario: Invalid VAT (1)
Given dataEntity VAT
When e_VAT.VATValue values are "0,08; 0,25"
Then message "Incorrect VAT values"

Scenario: Invalid VAT (2)
Given dataEntity VAT
When e_VAT.VATValue partitionClass is Integer
Then message "Incorrect VATValue PartitionClass"
```

Spec. 7. Example of a Gherkin (partial) specification of data entity tests.

```
UseCase uc_2_ManageCustomers "Manage Customers" : EntitiesManage [
    actorInitiates aU_Operator
    dataEntity ec_Customer
    actions aClose, aSearch, aFilter, aCreate, aRead, aUpdate, aDelete, aPrint_Customer
    extensionPoints Create, Read, Update, Delete, Print_Customer, Print_Customers]

UseCase uc_2_1_CreateCustomer "Create Customer" : EntityCreate [
    actorInitiates aU_Operator
    dataEntity ec_Customer
    actions aSave, aCancel
    extends uc_2_ManageCustomers onExtensionPoint Create]

UseCase uc_2_2_ConsultCustomer "Consult Customer" : EntityRead [

UseCase uc_2_3_UpdateCustomer "Update Customer" : EntityUpdate [

UseCase uc_2_4_DeleteCustomer "Delete Customer" : EntityDelete [
    actorInitiates aU_Operator
    dataEntity ec_Customer
    actions aDelete, aCancel
    extends uc_2_ManageCustomers onExtensionPoint Delete]
```

Spec. 8. Example of a TSL (partial) specification of Use Cases.

```
UseCaseTestCase t_uc_2_1_CreateCustomer "Create Customer" : Valid [
     useCase uc_2_1_CreateCustomer
     actorInitiates aU_Operator

     testScenario CreateCustomer_Generic :Main [
        isAbstract
        testStep s1:System_Execute:ShowData ["The System shows a CreateCustomer form"]
        testStep s2:Actor_PrepareData ["The Operator fills the fields Name and FiscalID"]
        testStep s3:Actor_CallSystem ["The Operator click the Save button"]
        testStep s4:System_Execute ["The System check if the Name and FiscalID are unique"]
        testStep s5:System_ReturnResult ["The System shows a successful notification message"] ]

    testScenario CreateCustomer_Concrete :Alternative [
        isConcrete
        testStep s1:System_Execute:ShowData ["The System shows a CreateCustomer form"]
        testStep s2:Actor_PrepareData ["The Operator fills the fields with the following data:"
                dataEntity e_Customer  withValues (
                |   e_Customer.Name |   e_Customer.fiscalID         +|
                |   "Mary White"     |   "999 997 688"             +|
                )]
        testStep s3:Actor_CallSystem ["The Operator click the Save button"]
        testStep s4:System_Execute ["The System check that Name and FiscalID are unique"]
        testStep s5:System_ReturnResult ["The System shows a successful notification message"] ]

    testScenario CreateCustomer_Exception :Exception [
        testStep s1:System_Execute:ShowData ["The System shows a CreateCustomer form"]
        testStep s2:Actor_PrepareData ["The Operator fills the fields Name and FiscalID"]
        testStep s3:Actor_CallSystem ["The Operator click the Save button"]
        testStep s4:System_Execute ["The System check if the Name and FiscalID are unique,
                                    and if they exist already the System does not save the data"]
        testStep s5:System_ReturnResult ["The System shows a error notification message!"] ]
]
```

Spec. 9. Example of a TSL (partial) specification of Use Case tests.

5.2 TSL: Use Case Tests

Use case tests are derived from various use cases (Spec. 8) expressed by RSL. TSL defines Use Cases Test Cases (Spec. 9) from the RSL system-level view *Actors* and *UseCases*. Each test contains multiple test scenarios, which are derived from the various flows of each use case. A test scenario encompasses of a group of test steps and shall be executed by an actor, which are also derived from the RSL.

The UseCaseTestCase 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.

For each scenario, it is specified a *name*, the scenario *Type* (Main, Alternative or Exception flow, respectively), and the *set of steps* involved. For each test step, it must be indicated the *actor* who performs it [Actor], a reference to the equivalent *use case step* [Step] if relevant, and an informal step *definition*, that describes the action executed. The equivalent Gherkin specification of the TSL Use Case tests is illustrated in Spec. 10.

```
Feature: Use Case: Create Customer

Scenario: CreateCustomer_Generic
        given "I'm in the role of Operator"
         when "I select the Create Customer option"
          and "The System shows a Create Customer form with the following fields:
               Name, FiscalID"
          and "I fill the fields Name and FiscalID"
          and "I click the Save button"
         then "The System check if the Name and FiscalID are unique, and saves the data"
          and "The System shows a successful notification message"

Scenario: CreateCustomer_Concrete
        given "I'm Peter Brown in the role of Operator"
         when "I select the Create Customer option"
          and "The System shows a Create Customer form with the following fields:
              Name, FiscalID"
          and "I fill the fields Name and FiscalID"
          and  dataEntity Customer with values:
                |   e_Customer.Name     |   e_Customer.FiscalID     |
                |   "Mary White"        |   "999 997 688"           |
          and "I click the Save button"
         then "The System check if the Name and FiscalID are unique, and saves the data"
          and "The System shows a successful notification message"

Scenario: CreateCustomer_Generic
        given "I'm in the role of Operator"
         when "I select the Create Customer option"
          and "The System shows a Create Customer form with the following fields: Name, FiscalID"
          and "I fill the fields Name and FiscalID"
          and "I click the Save button"
         then "The System check if the Name and FiscalID are unique, and if they exist already
               the System does not save the data"
          and "he System shows a error notification message: The customer data already exist!"
```

Spec. 10. Example of a Gherkin (partial) specification of Use Case tests.

5.3 TSL: State Machines Tests

Spec. 12 shows some examples of *StateMachineTestCase* associated to the *e_Invoices* s_m_e_invoice state machine (see Spec. 11).

```
StateMachine sm_e_Invoice "StateMachine_Invoice" : Complex [
    dataEntity e_Invoice
    description "StateMachine of entity Invoice"
    state StateInitial isInitial onEntry "In creation"
        useCase uc_1_1_CreateInvoice action aCreate nextState PendingState
    state PendingState onEntry "e.state= 'Pending'; e.isApproved= False"
    state ApprovedState onEntry "e.state= 'Approved'; e.isApproved= True"
    state RejectedState onEntry "e.state= 'Rejected'; e.isApproved= False"
    state PaidState isFinal onEntry "e.state= 'Paid'"
    state DeletedState isFinal onEntry "e.state= 'Deleted'"]
```

Spec. 11. Example of a TSL (partial) specification of state machine.

The first (i.e., *tsm1_SME_Invoice*) is an invalid test case because it defines an invalid sequence of states (namely, *Initial*, *Pending*, *Paid*). The second (i.e., *tsm2_SME_Invoice*) is a valid test case because it defines a valid sequence of states related with a reject situation (namely involving the following sequence of states: *Initial*, *Pending*, *Rejected*, *Deleted*, *Archive*); the third (i.e., *tsm3_SM_E_Invoice*) is also a valid test case but it defines a valid sequence of states related with an approved and paid situation.

```
StateMachineTestCase tsm1_SM_E_Invoice "tsm1_SM_E_Invoice Invalid" : Invalid [
    stateMachine sm_e_Invoice
    stateSequence PendingState, PaidState, RejectedState
    message "(SM_E_Invoice) Invalid State Sequence"]

StateMachineTestCase tsm2_SM_E_Invoice "tsm2_SM_E_Invoice Valid" : Valid [
    stateMachine sm_e_Invoice
    stateSequence PendingState, RejectedState, DeletedState
    message "(SM_E_Invoice) Valid State Sequence - Rejected Invoice"]

StateMachineTestCase tsm3_SM_E_Invoice "tsm3_SM_E_Invoice Valid" : Valid [
    stateMachine sm_e_Invoice
    stateSequence PendingState, ApprovedState, PaidState
    message "(SM_E_Invoice) Valid State Sequence - Approved Invoice"]
```

Spec. 12. Example of a TSL (partial) specification of state machine tests.

Spec. 13 shows the equivalent specification of these state machine test cases in the Gherkin language.

```
Feature: Management of Invoice data entity

Scenario: tsm1_SM_E_Invoice Invalid
Given stateMachine sm_e_Invoice
When stateSequence Initial > Pending > Paid > Rejected
Then Output "(SM_E_Invoice) Invalid State Sequence"

Scenario: tsm2_SM_E_Invoice Valid
Given stateMachine sm_e_Invoice
When stateSequence Initial > Pending > Rejected > Deleted> Archived
Then Output "(SM_E_Invoice) Valid State Sequence - Rejected Invoice"

Scenario: tsm3_SM_E_Invoice Valid
Given stateMachine sm_e_Invoice
When stateSequence Initial > Pending > Approved > Paid
Then Output "(SM_E_Invoice) Valid State Sequence - Approved Invoice"
```

Spec. 13. Example of a Gherkin (partial) specification of state machine tests.

6 Conclusion

This paper describes the TSL language and the companion model-based testing approach where test cases are derived from RSL constructs that describe the system behavior, such as *Actor* view, *DataEntity* view, *UseCase* view and *StateMachine* view. Based on a black-box approach and from these views, it is possible to define three main test constructs: data entity tests, state machine tests and use case tests

The approach is illustrated based on a fictitious informal specification of an invoice management application, the "Billing System". It shows how the test cases constructs may be represented and demonstrates that by using executable requirements specifications, functional tests can be easy to "read, write, execute, debug, validate, and maintain" [8].

As future work, we intend to explore and implement test case input data generation techniques (e.g., through domain analysis and resolution of constraints on attribute values) and implement other algorithms for generating test cases from state machines based on other coverage criteria, for example, Switch-1 or Switch-2. Also, we aim to extract test scenarios based on the varies flows expressed by Use Cases and extend the language to also support user story test cases. In addition, we aim to automate the TSL test case generation processes that may be based on the following transformations: generate TSL test cases from equivalent RSL requirements specifications; and directly from existent systems and databases, namely adopting model-driven reverse engineering techniques like we researched recently [13].

Finally, we aim to automate even further the whole process by using test frameworks like Cucumber or Specflow which enable to execute automatically test cases written in Gherkin.

References

1. Bettini, L., 2016. Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd.
2. Buuren, Robin A. ten., 2015. Domain-Specific Language Testing Framework. (October).
3. Fagan, M.E., 2001. Advances in software inspections. In Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Springer.
4. Ferreira, D., Silva, A.R., 2012. RSLingo: An information extraction approach toward formal requirements specifications, Proceedings of MoDRE2012, IEEE CS.
5. Ferreira, D., Silva, A.R., 2013. RSL-PL: A Linguistic Pattern Language for Documenting Software Requirements, in Proceedings of RePa13, IEEE CS.
6. Ferreira, D., Silva, A.R., 2013a. RSL-IL: An Interlingua for Formally Documenting Requirements, in Proc. of the of Third IEEE International Workshop on Model Driven Requirements Engineering, IEEE CS.
7. Ibe, M., 2013. Decomposition of test cases in model-based testing, in CEUR Workshop Proceedings.
8. King, T., 2014. Functional Testing with Domain- Specific Languages.
9. Kovitz, B., 1998. Practical Software Requirements: Manual of Content and Style. Manning.
10. Monteiro, T., Paiva, A.C.R., 2013. Pattern Based GUI Testing Modeling Environment, Sixth International Conference on Software Testing, Verification and Validation (ICST) Workshops Proceedings.
11. Morgado, I., Paiva, A.C.R., 2017. Mobile GUI testing, Software Quality Journal, pp.1-18.
12. Paiva, A.C.R., 1997. Automated Specification-based Testing of Graphical User Interfaces, Ph.D. thesis, Faculty of Engineering, Porto University, Porto, Portugal.
13. Reis, A., Silva, A.R., 2017. XIS-Reverse: A Model-Driven Reverse Engineering Approach for Legacy Information Systems, Proceedings of MODELSWARD2017, SCITEPRESS.
14. Ribeiro, A., Silva, A.R., 2014. XIS-Mobile: A DSL for Mobile Applications, Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC).
15. Ribeiro, A., Silva, A.R., 2014a. Evaluation of XIS-Mobile, a Domain Specific Language for Mobile Application Development, Journal of Software Engineering and Applications, 7(11), pp. 906-919.
16. Moreira, R.M.L.M., Paiva, A.C.R., Nabuco, M., and Memon, A., 2017. Pattern-based GUI testing: bridging the gap between design and quality assurance. Software Testing, Verification and Reliability Journal, 27(3).
17. Savic, D., et al, 2015. SilabMDD: A Use Case Model Driven Approach, ICIST 2015 5th International Conference on Information Society and Technology.
18. Silva, A.R., 2015. Model-Driven Engineering: A Survey Supported by a Unified Conceptual Model, Computer Languages, Systems & Structures 43 (C), 139155.
19. Silva, A.R., 2015. SpecQua: Towards a Framework for Requirements Specifications with Increased Quality, in Lecture Notes in Business Information Processing (LNBIP), LNBIP 227, Springer.
20. Silva, A.R., et al, 2015. A Pattern Language for Use Cases Specification, in Proceedings of EuroPLOP2015, ACM.
21. Silva, A.R., Saraiva, J., Ferreira, D., Silva, R., Videira, C., 2007. Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools, IET Software, IET.

22. Silva, A.R., Saraiva, J., Silva, R., Martins, C., 2007. XIS UML Profile for eXtreme Modeling Interactive Systems, in Proceedings of MOMPES'2007, IEEE Computer Society.
23. Silva, A.R., Verelst, J., Mannaert, H., Ferreira, D., Huysmans, P., 2014. Towards a System Requirements Specification Template that Minimizes Combinatorial Effects, Proceedings of QUATIC2014 Conference, IEEE CS.
24. Silva, A.R., 2017. Linguistic Patterns and Linguistic Styles for Requirements Specification (I): An Application Case with the Rigorous RSL/Business-Level Language, in Proceedings of EuroPLOP2017, ACM.
25. Silva, A.R., 2017. A Rigorous Requirement Specification Language for Information Systems: Focus on RSLs Use Cases, Data Entities and State Machines, INESC-ID Technical Report.
26. Solis, C., & Wang, X., 2011. A study of the characteristics of behaviour driven development. In Software Engineering and Advanced Applications (SEAA), 37th EUROMICRO Conference on (pp. 383-387). IEEE.
27. Stahl, T., Volter, M., 2005. Model-Driven Software Development, Wiley.
28. Verelst, J., Silva, A.R., Mannaert, H., Ferreira, D., Huysmans, 2013. Identifying Combinatorial Effects in Requirements Engineering. In Proceedings of Third Enterprise Engineering Working Conference (EEWC 2013), Advances in Enterprise Engineering, LNBIP, Springer.
29. Videira, C., Silva, A.R., 2005. Patterns and metamodel for a natural-language-based requirements specification language. CAiSE Short Paper Proceedings.
30. Videira, C., Ferreira, D., Silva, A.R., 2006. A linguistic patterns approach for requirements specification. Proceedings 32nd Euromicro Conference on Software Engineering and Advanced Applications (Euromicro'2006), IEEE Computer Society.
31. Bhat, A., Quadri, S.M.K., 2015. Equivalence class partitioning and boundary value analysis - A review. 2nd International Conference on Computing for Sustainable Global Development (INDIACom).
32. Silva, A.R., Paiva, A.C.R., Silva, V.E.R., 2018. Towards a Test Specification Language for Information Systems: Focus on Data Entity and State Machine Tests, in the 6th International Conference on Model-Driven Engineering and Software Development.
33. Jacobson, I., et al., 1992. Object oriented Software engineering: A Use Case Driven Approach, Addison- Wesley.
34. Silva, A.R., 2017. RSLingo RSL Excel Template, v4.0, October 2017, Available at www.researchgate.net/publication/320256323_RSLingo_RSL_Excel_Template_v40.
35. Neto, A.C.D., Subramanyan, R., Vieira, M., and Travassos, G.H., 2007. A survey on model-based testing approaches: a systematic review. Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering.
36. Hasling, B., Goetz. H., Beetz, K., 2008. Model based testing of system requirements using UML use case models. 1st international conference on IEEE Software Testing, Verification, and Validation.
37. Utting, M., Pretschner, A., Legeard, B., 2012. A taxonomy of modelbased testing approaches. Software Testing, Verification and Reliability 22.5 (2012): 297-312.
38. Manfred, B., Jonsson, B., Katoen, J., Leucker, M., Pretschneret, A., 2005. Model-based testing of reactive systems. Volume 3472 of Springer LNCS.
39. Wynne, M., Hellesoy, A., & Tooke, S., 2017. The cucumber book: behaviour-driven development for testers and developers. Pragmatic Bookshelf.

40. Cucumber & Gherkin, Accessed in March 2018, https://cucumber.io/docs/reference
41. Paiva, A.C.R., Vilela, L., 2017. Multidimensional test coverage analysis: PARADIGM-COV tool. Cluster Computing - The Journal of Networks Software Tools and Applications, Volume 20, pp.633-649.
42. Nabuco, M., Paiva, A.C.R., 2014. Model-Based Test Case Generation for Web Applications. In 14th International Conference Computational Science and Its Applications - ICCSA.