Towards a Test Specification Language for Information Systems: Focus on Data Entity and State Machine Tests

Alberto Rodrigues da Silva[1], Ana C. R. Paiva[2,3] and Valter Emanuel R. da Silva[2]
[1]*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal*
[2]*Faculty of Engineering of the University of Porto, Porto, Portugal*
[3]*INESC TEC, Porto, Portugal*

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

Abstract. This paper introduces the TSL language (short name for "Test Specification Language") that intends to improve the test specification of information systems in a systematic, rigorous and consistent way. TSL specifications are produced from close requirement specifications expressed in the RSL language (Requirements Specification Language). Both RSL and TSL support human-readable executable specifications closer to natural language than models usually used in model-based testing approaches. TSL includes several constructs logically arranged into views according to multiple testing engineering strategies, commonly found in the information systems domain, such as: data entity tests and state machine tests, all of them produced from equivalent requirement specification in RSL. A case study is also presented to illustrate the proposed approach.

1 INTRODUCTION

Testing is one of the most important activities to ensure the quality of a software system in the scope of software development projects. As reported by Ibe, about 30 to 60 percent of the total effort within a project is spent on testing (Ibe, 2013). It is also estimated that up to 50 percent of the total development costs are related to testing (Fagan, 2001). This indicates, not only its importance, but also the higher impact it has in the overall system development process cycle.

Model-based testing (MBT) is one technique that addresses this problem (Stahl and Volter, 2005; Silva, 2015; Morgado, 2017). A potential infinite set of test cases can be generated from a model of a given "system under test" (SUT) (or just "system" for brevity). System models or system specifications vary in nature: they can be more or less abstract and represented textually (Paiva, 1997) and/or graphically (Monteiro, 2013); they can describe the functionalities or goals (Rodrigo, 2017) of the system under test. A problem is that often these models do not exist, which demand they have to be developed from scratch, or there is only a textual description of its requirements with a very informal way, which does not allow to derivate automatically test cases from it. However, the existence of system requirements specification (SRS), defined with controlled natural languages, may enable the derivation of test cases directly from such rigorous models or specifications.

Usually system tests and acceptance tests (like requirement specifications) are manually written in some natural language. However, the resultant test cases are ineffective since they are hard to write and costly to maintain. 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 concerning the adoption of DSLs for test specification, namely (Buuren, 2015): (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 presents and discusses the TSL (Test Specification Language) that adopts a model-

Silva, A., Paiva, A. and Silva, V.
Towards a Test Specification Language for Information Systems: Focus on Data Entity and State Machine Tests.
DOI: 10.5220/0006608002130224
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 213-224
ISBN: 978-989-758-283-7

based testing approach for rigorous and human-readable specification of test cases. TSL is strongly inspired on the grammar, nomenclature and writing style as defined by the RSL, which is a rigorous requirements specification language (Silva, 2017; Silva, 2017a). By applying black-box functional testing design techniques, TSL includes and supports two different test strategies, namely, (i) domain analysis testing (the test strategy uses techniques such as equivalence partitioning and boundary value analysis for the definition of structural data values); (ii) state machine testing (the test strategy traverses the State Machine expressed in RSL according to different coverage criteria, e.g., cover all states).

To better support the explanation and discussion of the TSL language we introduce a fictitious information system (the "BillingSystem") that is partially described as a variety of informal requirements such as the following text. This description is to some extent deliberately incomplete, vague and inconsistent, as it is common in real-world situations.

Informal Requirements of a 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 the 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 […].

This paper is organized in 5 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 around Cucumber and Gherkin. Section 4 presents and discusses the TSL constructs and views, namely tests based on data entities and state machines. Finally, Section 5 presents the conclusion and identifies issues for future work.

2 RSL OVERVIEW

RSLingo is a long-term research initiative in the RE (Requirements Engineering) area that recognizes that natural language, although being the most common and preferred form of representation used within requirements documents, it is prone to produce such ambiguous and inconsistent documents that are hard to automatically validate or transform. Originally RSLingo 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 (Ferreira and Silva, 2012). This was achieved through the use of two original languages: the RSL-PL (Pattern Language) (Ferreira and Silva, 2013), designed for encoding RE-specific linguistic patterns, and RSL-IL (Intermediate Language), a domain specific language designed to address RE concerns (Ferreira and Silva, 2013a). Through the use of these two languages and the mapping between them, the initial knowledge written in natural language can be extracted, parsed and converted to a more structured format, reducing its original ambiguity and creating a more rigorous SRS document (Silva, 2015a).

More recently, Silva et al. designed a broader and more consistent language, called "RSLingo's RSL" (or just "RSL" for the sake of brevity), based on the design of former languages (Videira and Silva, 2005; Videira et al, 2006; Silva et al, 2007; Ferreira and Silva, 2013; Ferreira and Silva, 2013a; Ribeiro and Silva, 2014; Ribeiro and Silva, 2014/a; Silva et al, 2015; Savic et al, 2015). According to its authors RSL is a control natural language to help the production of SRSs in a systematic, rigorous and consistent way (Silva, 2017; Silva, 2017a). 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. As summarized in Table 1, these views are organized according to two abstraction levels: business and system levels; and to five concerns: context, active structure, behaviour, passive structure and requirements.

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 behaviour; (3) the common terms used in that business 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 comprise respectively the following views: *Stakeholders* (active structure concern), *BusinessProcesses* (behaviour 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: (1) constructs that allow to describe the actors that interact with the system; (2) constructs that allow to describe the behaviour of some system's data entities, namely based on state machines; (3) constructs that allow to describe the structure of the system, namely

Table 1: Classification of RSL views: abstraction levels versus RE specific concerns (Silva, 2017).

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

```
/**********************************************************
   System definition
**********************************************************/
system   S_billingsystem : System [
  name "BillingSystem"
  scope In
  subSystems S_billingsystem, S_billingsystem
  description ""
]

/**********************************************************
   Entities
**********************************************************/
dataEntity e_VAT : Principal [
  name "VAT"
  attribute VATCode: Integer [name "VAT Code" NotNull Unique]
  attribute VATName: String(30) [name "VAT Class Name" NotNull]
  attribute VATValue: Decimal(2.1) [name "VAT Class Value " NotNull]
  primaryKey (VATCode)
  description "VAT Class"]

dataEntity e_Product : Principal [
  name "Product"
  attribute ID: Integer [name "Product ID" NotNull Unique]
  attribute Name: String(50) [name "Name" multiplicity "1..2" description "Product Name"]
  attribute valueWithoutVAT: Decimal(16.2) [name "Price Without VAT" NotNull ]
  attribute valueWithVAT: Decimal(16.2) [name "Price With VAT" NotNull ]
  attribute VATCode : Integer [name "VAT Code" NotNull]
```

Specification 1: RSL (partial) specification example.

based on data entities and data entity views; and (4) constructs that allow 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).

Specification 1 shows a simple text snippet of the RSL requirements specification for the BillingSystem example.

3 GHERKIN/CUCUMBER OVERVIEW

Behavior-driven Development (BDD) is a software development methodology in which an application is specified and designed describing how its beha-vior should appear to an external observer (Solis and Wang, 2011). In BDD, people like business analysts or product owners first write acceptance tests that describe the system behavior from the user's point of view. Then these acceptance tests shall be reviewed and approved by product owners before developers start write their software code.

Cucumber[1] is a test tool that executes automated acceptance tests written in a behaviour-driven style (BDD). Cucumber enables automation of functional validation in an easily readable and understandable format (as plain English) for business analysts, developers, testers, and others.

Gherkin[2] is a popular language used by Cucumber to define test cases. Its main objective is to enable to specify tests in a way that clients can understand them. 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 below, more information can be obtained, for example, in [1].

Simple test case example in Gherkin:

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*

4 TSL APPROACH AND LANGUAGE

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

Figure 1 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 Figure 2, a TSL specification is a combination of two different types of elements. First, the *TestSupportSpecs* package includes *TestSupportSpec* elements such as *DataEntities* or *StateMac hines*. 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* or *StateMachineTestCase*. Each *TestCase* shall be defined as Valid or Invalid and shall have a dependency to a respective *TestSupportSpec*, e.g., a *StateMachineTestCase* shall have a dependency to the respective *StateMachine*. These *TestCase*

[1] https://cucumber.io/

[2] https://cucumber.io/docs/reference#gherkin

elements can be generated by the RSL-to-TSL and TSL-to-TSL transformations, but usually shall be 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 (Bettini, 2016). 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 is extended with the creation of three 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 the Xtext format.

As suggested above in Figure 2, TSL supports the specification of different test generation techniques from RSL specifications, namely *DataEntity*, *StateMachine* and *UseCase* test cases.

DataEntityTestCases can be defined by applying equivalence class partitioning and boundary value analysis (Bhat and Quadri, 2015) over RSL *DataEntities*. On the other hand, *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. Furthermore, *UseCaseTestCase* can be defined by exploring multiple sequences of steps defined in RSL use cases, and also by associating data values to the involved data entities. Due to space constraints we do not show in this paper the part of the TSL language related to

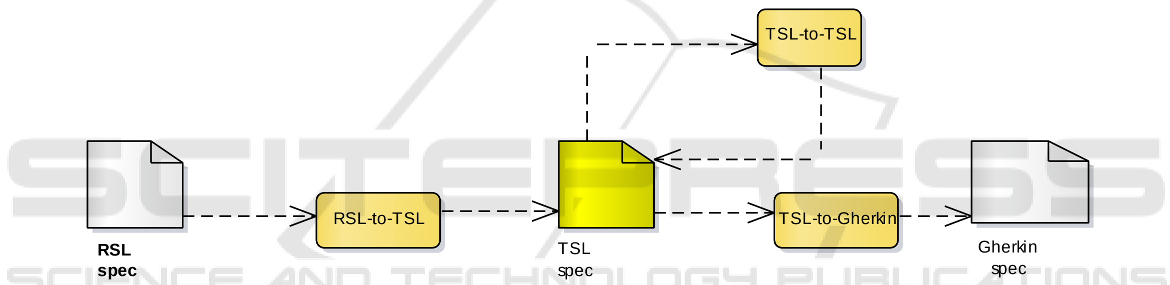

Figure 1: TSL based approach.

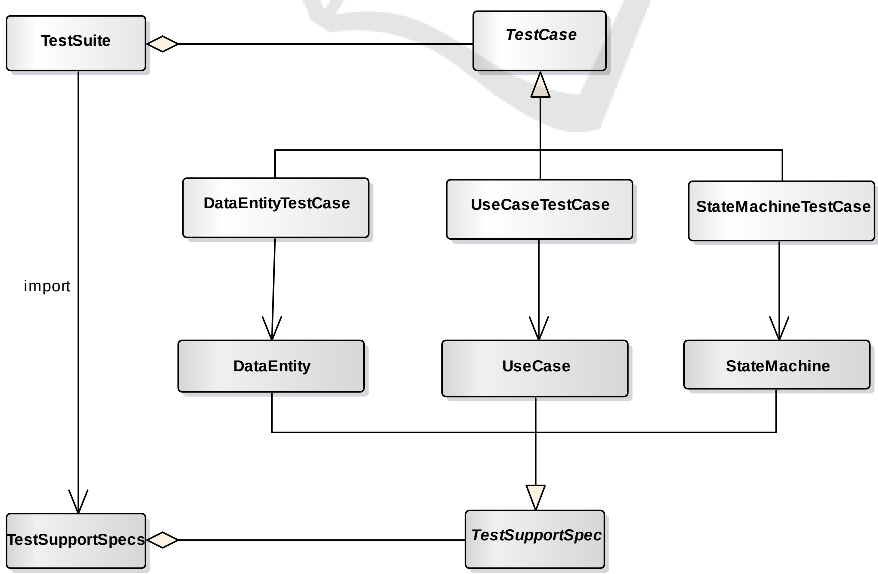

Figure 2: Metamodel of the TSL general architecture (partial view).

```
package-tests billingsystem.testspecs

testSupportSpecs tss_billing : SystemTestSupportSpecs [ name "BillingSystem - Test Support Elements"]

/**********************************************************
   Entities
**********************************************************/

dataEntity e_VAT : Principal [
    name "VAT"
    attribute VATCode: Integer [name "VAT Code" NotNull Unique]
    attribute VATName: String(30) [name "VAT Class Name" NotNull]
    attribute VATValue: Decimal(2.2) [name "VAT Class Value " NotNull]
    primaryKey (VATCode)
    description "VAT Class"]

dataEntity e_Product : Principal [
    name "Product"
    attribute ID: Integer [name "Product ID" NotNull Unique]
    attribute Name: String(50) [name "Name" multiplicity "1..2" description "Product Name"]
    attribute valueWithoutVAT: Decimal(16.2) [name "Price Without VAT" NotNull ]
    attribute valueWithVAT: Decimal(16.2) [name "Price With VAT" NotNull ]
    attribute VATClassCode : Integer [name "VAT Code" NotNull]
    attribute VATClassValue : Decimal(2.2) [name "VAT Class Value " NotNull]
    primaryKey (ID)
    foreignKey e_VAT(VATClassCode)
    description "Products"]

/**********************************************************
   Actors
**********************************************************/
actor aU_Operator : User [name "Operator" description "Operator user"]
actor aU_Manager : User [name "Manager" description "Manager user"]
actor aU_TechnicalAdmin : User [name "TechnicalAdmin"]
actor aU_Customer : User [name "Customer"]


/**********************************************************
   Use Cases
**********************************************************/

useCase uc_1_ManageInvoices: EntitiesManage [
  name "Manage Invoices"
 actorInitiates  aU_Operator
  actions Close, Search, Filter, Create, Read, Update, Send_Invoice, Export_Invoices, Pri
  extensionPoints EPCreate, EPRead, EPUpdate, EPConfirmPayment, EPDelete, EPSendInvoices,
  description "Manage Invoices"
]
/**********************************************************
   StateMachines
**********************************************************/

stateMachine sm_e_Invoice: Complex [
  name "Invoice_StateMachine"
  dataEntity e_Invoice
  state Initial isInitial onEntry "Invoice e= new Invoice();"
  state Pending onEntry "e.state= 'Pending'; e.isApproved= False;"
  state Approved onEntry "e.state= 'Approved'; e.isApproved= True;"
  state Rejected onEntry "e.state= 'Rejected'"
  state Paid onEntry "e.state= 'Paid'"
  state Deleted onEntry "e.state= 'Deleted'"
  state Archived isFinal onEntry "e.state= 'Archived'"
]
```

Specification 2: Example of a TSL (partial) specification defined in a TestSupportSpecs package.

UseCaseTestCases (that shall be discussed in a future work).

4.1 Test Support Specs

As referred above, a *TestSupportSpecs* package defines the supported elements that shall be then used by the test cases. Specification 2 shows a TSL partial specification of a *TestSupportSpecs* package for the BillingSystem example. In particular it shows the specification of the following elements: *e_VAT* and *e_Product* data entities, actors, the *uc_1_ManageInvoices* use case, and the *sm_e_Invoice* state machine.

4.2 Data Entity Test Cases

Domain analysis testing is based on classic test design techniques known as "equivalence class partitioning" and "boundary value analysis" (Bhat and Quadri, 2015). Since most of the times it is unfeasible to test all possible values of the possible domain classes or data entities (in the RSL/TSL terminology), the equivalence class partitioning technique partitions the domain into equivalent classes (assuming that the behaviour of the system is the same for every value of a class) and then tests one value for each class. For boundary value analysis, the input values are the ones located at the boundaries of the equivalence classes because it is expected that the probability of finding failures is higher.

As shown in Specification 2, 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. According to equivalence class partitioning we should test valid and invalid input values. So, for this particular case, the tester could define a valid input, e.g., 15.2, and an invalid input, e.g., 14.35. Of course, the tester can opt to define an invalid input value according to the type of the attribute. In this case a possible invalid value would be, for example, a string, e.g., "invValue".

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, if we have an attribute B that ranges from 5 to 7, the tester can define test input data on the boundaries, e.g., 5 and 7 for valid, and 4 and 8 for invalid values.

As illustrated in Figure 3, 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.

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. Each product has a price with and without the respective VAT. The VAT (value-added tax) VAT 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 3 shows a TSL specification of some of these entities, namely the *e_VAT* and *e_Product* data entities.

Based on this data entities specification it is possible to define and also to generate some data entity test cases. Specification 4 shows some of these tests defined for the *e_VAT* data entity. First, *detVAT1* is defined as a valid test case 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*").

Specification 5 shows the equivalent data entity test case in the Gherkin language.

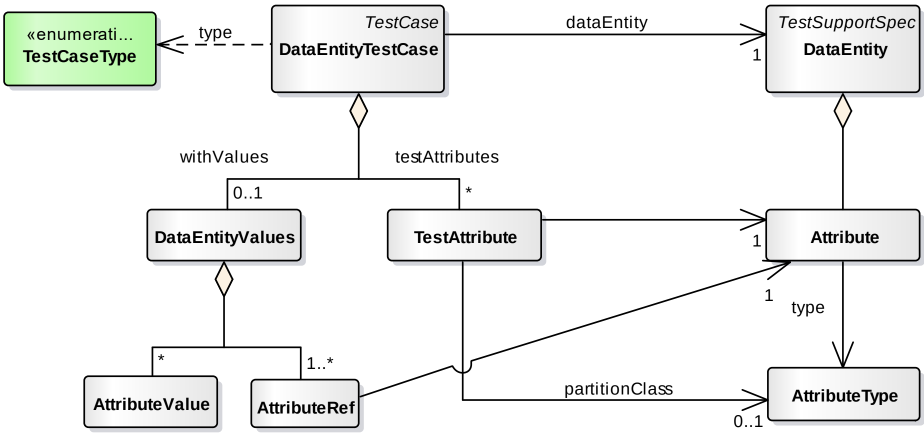

Figure 3: Metamodel of the TSL DataEntityTestCase definition (partial view).

```
dataEntity e_VAT : Principal [
    name "VAT"
    attribute VATCode: Integer [name "VAT Code" NotNull Unique]
    attribute VATName: String(30) [name "VAT Class Name" NotNull]
    attribute VATValue: Decimal(2.2) [name "VAT Class Value " NotNull]
    primaryKey (VATCode)
    description "VAT Class"]

dataEntity e_Product : Principal [
    name "Product"
    attribute ID: Integer [name "Product ID" NotNull Unique]
    attribute Name: String(50) [name "Name" multiplicity "1..2" description "Product Name"]
    attribute valueWithoutVAT: Decimal(16.2) [name "Price Without VAT" NotNull ]
    attribute valueWithVAT: Decimal(16.2) [name "Price With VAT" NotNull ]
    attribute VATClassCode : Integer [name "VAT Code" NotNull]
    attribute VATClassValue : Decimal(2.2) [name "VAT Class Value " NotNull]
    primaryKey (ID)
    foreignKey e_VAT(VATClassCode)
    description "Products"]
```

Specification 3: Example of a TSL (partial) specification of data entities.

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

dataEntityTestCase detVAT2: Valid [ name "detVAT2"
    dataEntity e_VAT  withValues (
    |   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
    )
    testAttribute  e_VAT.VATCode (partitionClass Integer)
    testAttribute  e_VAT.VATName (partitionClass String)
    testAttribute e_VAT.VATValue (partitionClass Decimal(2.2))

    message "Correct VAT values (2)"
]

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

Specification 4: Example of a TSL (partial) specification of data entity tests.

```
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 Valid VAT (1)

Scenario outline: 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 VAT Valid (2)


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

Specification 5: Example of a Gherkin (partial) specification of data entity tests.

4.3 State Machine Test Cases

A state machine is a model that describes the dynamic behaviour of a system over a given data entity (or object) throughout its life-cycle. A state machine allows to represent the behaviour 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 the 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.

As illustrated in Figure 4, 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.

The Specification 6 shows some examples of *StateMachineTestCase* associated to the *e_Invoice*'s state machine (as previously defined above, shown in Specification 2). The first (i.e., *tsm1_SM_E_Invoice*) is an invalid test case because it defines an invalid sequence of states (i.e., Initial, Pending, Paid); The second (i.e., *tsm2_SM_E_Invoice*) is a valid test case because it defines a valid sequence of states related with a reject situation (i.e., Initial, Pending, Rejected, Deleted, Archive); the third (i.e., *tsm3_SM_E_Invoice*) is also a valid test case because it defines a valid sequence of states related with an approved and paid situation.

Specification 7 shows the equivalent state machine test case in the Gherkin language.

5 CONCLUSION

This paper describes the Test Specification Language (TSL), a model-based test 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 behaviour, such as *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: data entity tests, state machine tests and use case tests.

The study case "Billing System", a fictitious invoice management application, allowed to illus-

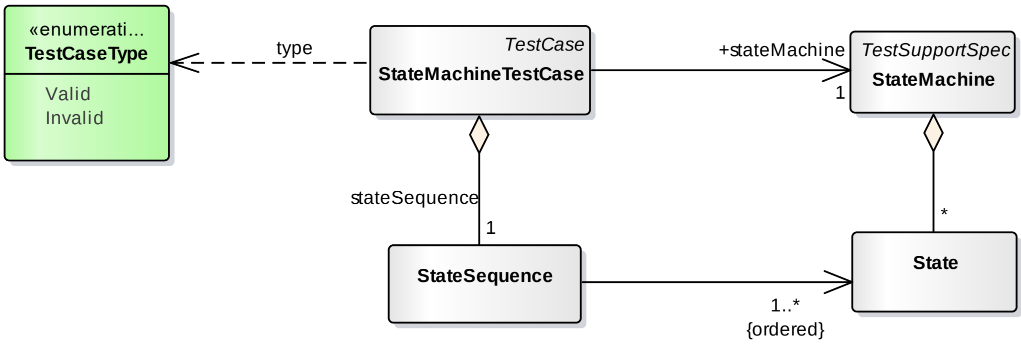

Figure 4: Metamodel of the TSL StateMachineTestCase definition (partial view).

```
/*********************************************************
   stateMachineTestCase
*********************************************************/

stateMachineTestCase tsm1_SM_E_Invoice : Invalid [ name "tsm1_SM_E_Invoice Invalid"
    stateMachine sm_e_Invoice
    stateSequence sm_e_Invoice.Initial, sm_e_Invoice.Pending, sm_e_Invoice.Paid
    message "(SM_E_Invoice) Invalid State Sequence"]

stateMachineTestCase tsm2_SM_E_Invoice : Valid [ name "tsm2_SM_E_Invoice Valid"
    stateMachine sm_e_Invoice
    stateSequence   sm_e_Invoice.Initial, sm_e_Invoice.Pending, sm_e_Invoice.Rejected,
            sm_e_Invoice.Deleted, sm_e_Invoice.Archived
    message "(SM_E_Invoice) Valid State Sequence - Rejected Invoice"]

stateMachineTestCase tsm3_SM_E_Invoice : Valid [ name "tsm3_SM_E_Invoice Valid"
    stateMachine sm_e_Invoice
    stateSequence   sm_e_Invoice.Initial, sm_e_Invoice.Pending, sm_e_Invoice.Approved,
            sm_e_Invoice.Paid, sm_e_Invoice.Archived
    message "(SM_E_Invoice) Valid State Sequence - Approved Invoice"]
```


Specification 6: Example of a TSL (partial) specification of state machine tests.

```
Feature: Management of VAT data entity

Scenario: Invalid State Sequence (1)
Given dataEntity VAT
When stateSequence Initial > Pending > Paid
Then Output "(SM_E_Invoice) Invalid State Sequence"

Scenario: Valid State Sequence (1)
Given dataEntity VAT
When stateSequence Initial > Pending > Rejected > Deleted> Archived
Then Output "(SM_E_Invoice) Valid State Sequence - Rejected Invoice"

Scenario: Valid State Sequence (2)
Given dataEntity VAT
When stateSequence Initial > Pending > Approved > Paid > Archived
Then Output "(SM_E_Invoice) Valid State Sequence - Approved Invoice"
```

Specification 7: Example of a Gherkin (partial) specification of state machine tests.

trate 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" (King, 2014).

As future work it shall be important to extend the language to support, in addition, both use case and user story test cases. It shall also be relevant to automate processes for TSL test case generation and we consider the following transformations: generate TSL test cases from equivalent RSL requirements specifications; and directly from existant systems and databases, namely adopting model-driven reverse engineering techniques like we researched recently (Reis and Silva, 2017). Futhermore, it shall be important the automatic execution of tests namely with their integration with external test frameworks.

At this point in time, 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[3] or Specflow[4] 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, Specflow 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.

ACKNOWLEDGEMENTS

This work was partially supported by national funds under FCT projects UID/CEC/50021/2013.

REFERENCES

Bettini, L., 2016. Implementing Domain-Specific Languages with Xtext and Xtend. *Packt Publishing Ltd*.

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

Fagan, Michael E., 2001. Advances in software inspections. In *Pioneers and Their Contributions to Software* Engineering: sd&m *Conference on Software Pioneers, Springer*.

Ferreira, D., Silva, A. R., 2012. RSLingo: An information extraction approach toward formal requirements specifications, *Proceedings of MoDRE'2012, IEEE CS*.

Ferreira, D., Silva, A. R., 2013. RSL-PL: A Linguistic Pattern Language for Documenting Software Requirements, in *Proceedings of RePa'13, IEEE CS*.

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.*

Ibe, Marcel, 2013. Decomposition of test cases in model-based testing, in *CEUR Workshop Proceedings*.

King, Tariq, 2014. Functional Testing with Domain-Specific Languages.

Kovitz, B. 1998. Practical Software Requirements: Manual of Content and Style. Manning.

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*.

Morgado, I., Paiva, A.C.R., 2017. Mobile GUI testing, *Software Quality Journal, pp.1-18*.

Paiva, A.C.R. (1997). Automated Specification-based Testing of Graphical User Interfaces, Ph.D. thesis, Faculty of Engineering, *Porto University, Porto, Portugal.*

Reis, A., Silva, A. R., 2017. XIS-Reverse: A Model-Driven Reverse Engineering Approach for Legacy Information Systems, *Proceedings of MODELSWARD'2017, SCITEPRESS*.

Ribeiro, A., Silva, A. R., 2014. XIS-Mobile: A DSL for Mobile Applications, *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*.

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*.

Rodrigo M. L. M. Moreira, Ana C. R. Paiva, Miguel Nabuco, and Atif Memon, 2017. Pattern-based GUI testing: bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability Journal, 27(3):e1629–n/a*.

Savic, D., et al, 2015. SilabMDD: A Use Case Model Driven Approach, *ICIST 2015 5th International Conference on Information Society and Technology*.

Silva, A. R., 2015. Model-Driven Engineering: A Survey Supported by a Unified Conceptual Model, *Computer Languages, Systems & Structures 43 (C), 139–155*.

Silva, A. R., 2015a. SpecQua: Towards a Framework for Requirements Specifications with Increased Quality, in Lecture Notes in *Business Information Processing (LNBIP), LNBIP 227, Springer*.

Silva, A. R., et al, 2015. A Pattern Language for Use Cases Specification, in *Proceedings of EuroPLOP'2015, ACM*.

[3] https://cucumber.io/

[4] http://specflow.org/

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*.

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*.

Silva, A. R., Verelst, J., Mannaert, H., Ferreira, D., Huysmans, P., 2014. Towards a System Requirements Specification Template that Minimizes Combinatorial Effects, *Proceedings of QUATIC'2014 Conference, IEEE CS*.

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 EuroPLOP'2017, ACM*.

Silva, A. R., 2017a. A Rigorous Requirement Specification Language for Information Systems: Focus on RSL's Use Cases, Data Entities and State Machines, *INESC-ID Technical Report*.

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*.

Stahl, T., Volter, M., 2005. Model-Driven Software Development, *Wiley*.

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*.

Videira, C., Silva, A. R., 2005. Patterns and metamodel for a natural-language-based requirements specification language. *CAiSE Short Paper Proceedings*.

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*.

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).*