Rigorous Specification of Use Cases with the RSL Language

Alberto Rodrigues da Silva
INESC-ID, Instítuto Superior Técnico,
Universidade de Lisboa
Lisboa, Portugal *alberto.silva@tecnico.ulisboa.pt*

Abstract

RSL language supports the specification of requirements in a systematic, rigorous and consistent way. RSL includes a large set of constructs to produce requirements specifications at different level of abstraction, different writing styles and different types of requirements (e.g., goals, functional requirements, quality requirements, constraints, user stories, and use cases) and tests. This paper focuses only on the RSL views related with use cases, including those constructs directly relevant to the specification of data-intensive information systems, namely: actors, use cases, data entities, state machines, and their respective relationships. The explanation and discussion is held by an illustrative example that shows how to produce such specifications. RSL offers an innovative approach that improves the way requirements specifications are defined and validated. In spite of other proposals, RSL is the first that integrates a large number of inter-related constructs that can be represented in a consistent and systematic way.

Keywords: RSL, Requirements Specification, Use Cases, Business Information Systems.

1. Introduction

Information systems involve the integration of multiple elements such as hardware, software apps, databases, organizational procedures and people to support its operations and processes. Requirements engineering (RE) is a discipline that provides a shared vision and understanding of the system under study among the involved stakeholders and throughout its life-cycle [16,28]. The negative consequences of ignoring these early RE activities are extensively reported and discussed in the literature [5,30,31].

A *system requirements specification* (SRS, or just "requirements specification") is an important document that helps to structure the concerns of such systems from the RE perspective. A good SRS offers several benefits as reported in the literature such as [4, 10,11,18,33]: contribute to the establishment of an agreement and business contract between customers and suppliers; provide a common ground for supporting the project's budget and schedule estimation and plan; support the project scope's validation and verification; and support all system deployment and future maintenance activities. It is usually recommended that a SRS shall be defined accordingly to a previously defined SRS template as well as a set of recommendations on how to customize and use it. A SRS template prescribes a given document structure with supplementary practical guidelines. In general these templates recommend the use of various views and constructs (e.g., actors, use cases, user stories) that might be considered "modular artefacts" in the sense of their definition and reuse. Because there are many dependencies among these constructs some authors argue that it is important to minimize or prevent them and some templates give some support in this respect [31]. On the other hand, and because SRSs are commonly used by both technical and business stakeholders, they tend to be specified mostly in *natural languages* to reach an effective communication level: everyone is able to communicate by means of a natural language because they are flexible, universal, and humans are proficient at using it to communicate with. So, natural languages have minimal adoption resistance as a requirements documentation technique. On the other hand, they exhibit some intrinsic features that frequently put themselves as the source of many quality problems, such as incorrectness, inconsistency, incompleteness and ambiguousness [10,11,16,18]. To mitigate such problems we found practical recommendations for better writing requirements including guidelines such as [7,16]: the language shall be used in a simple, clear and precise way, and shall follow a standardized format to give coherence and uniformity to all sentences; the sentences shall be short, simple and written in an affirmative and active voice style; or the vocabulary shall be limited. These recommendations are usually better supported by the adoption of *controlled natural languages (CNLs)* that define a restricted use of a natural language's grammar (syntax) and a set of terms (including the semantics of these terms) to be used in the restricted grammar [12,16]. The adoption of CNLs has the following advantages: their sentences are easy to understand since they are similar to sentences in natural language; they are less ambiguous than expressions in natural language, since they have a simplified grammar and a predefined vocabulary with a precise semantics; and they may be semantically verifiable and computational manipulated, since they have a formal grammar and a predefined set of terms.

We have designed a broader and consistent language, called "RSLingo RSL" (or just "RSL" for brevity), based on the design of former experiments and languages like ProjectIT-RSL [23,32], RSL-IL [7], RSL-IL4Privacy [2,22], XIS* [17,24], but also inspired on other proposals, such as Pohl Framework [15], SilabREQ [19] and SilabMDD [20]. *RSL is a controlled natural language that helps the production of both requirement and test specifications in a systematic, rigorous and consistent way* [13,25,27]. RSL includes a rich set of constructs logically arranged into views according to specific concerns that exist at different abstraction levels. These constructs are defined as linguistic patterns and represented textually by mandatory and optional fragments (text snippets). Conceptually RSL is a process- and tool-independent language, meaning it can be used and be adapted by multiple users and organizations with different processes/methodologies and supported by multiple types of software tools. However, in practice, RSL has been implemented with the Xtext framework (https://eclipse.org/Xtext/) in an Eclipse-based tool called "ITLingo-Studio" so, its specifications are rigorously defined and can be automatically validated and transformed into multiple representations and formats. A lightweight tool support is also provided based on a former RSL Excel template publicly available at github (https://github.com/ITLingo/RSL).

This paper introduces in Section 2 the RSL and its general architecture but focuses the presentation and discussion on just its constructs most related with *use cases approaches*, i.e., focused on the following constructs: *use cases, actors, data entities, state machines, and their inherent relationships*. In Section 3, the explanation of the language is supported by a fictitious information system, which helps to describe it and to show and discuss both its textual (in RSL concrete syntax) and graphical (in UML-like) representations. Section 4 presents and discusses the related work by comparing RSL with other languages and technologies. Finally, Section 5 presents the conclusion and identifies future work.

2. RSL Language and Focus on its Use Case related Constructs

RSL provides several constructs logically classified according to two dimensions (see Table 1): abstraction level and specific concerns they address. The abstraction levels are: business, application, software and hardware levels. The concerns are: active structure (subjects), behaviour (actions), passive structure (objects), requirements, tests, other concerns, and relations & sets. From a syntactical perspective, any construct can be used in any type of system regardless of its abstraction level. That means, for example, that it is possible to use a DataEntity construct at Application or SoftwareSystem levels but also at Business or even HardwareSystem levels. However, the use of a DataEntity at Business level shall be more general and incomplete in comparison with its use at Application or SoftwareSystem levels, that shall be more detailed. In addition, while some constructs (e.g., Stakeholder, ActiveElement, GlossaryTerm) are naturally applied to different types of systems, others are not so obviously applied (e.g., UseCase and Actor shall be just applied to Applications or SoftwareSystems).

Figure 1 shows the RSL partial metamodel that involves the definition of views with the respective constructs and inherent relations. The names of the relations illustrated in Figure 1 suggest their respective semantics: for example the "isA" reflexive relation between Actors means that actors might have generalization/specialization relations. When not explicitly expressed the name of a relation means by default that the source construct "uses" or "depends on" the target construct. For example, as illustrated in Figure 1, the StateMachine depends on the DataEntity and the UseCase constructs. As said above, this paper focuses only on use cases and their associated constructs, as suggested informally in Figure 2. So, to guide the explanation and discussion below we first introduce DataEntities and DataEntityClusters, second Actors, then UseCases, and finally StateMachines. These construct rules are formally defined with the Xtext framework [1].

Table 1. Classification of RSL constructs: abstraction levels versus specific concerns

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

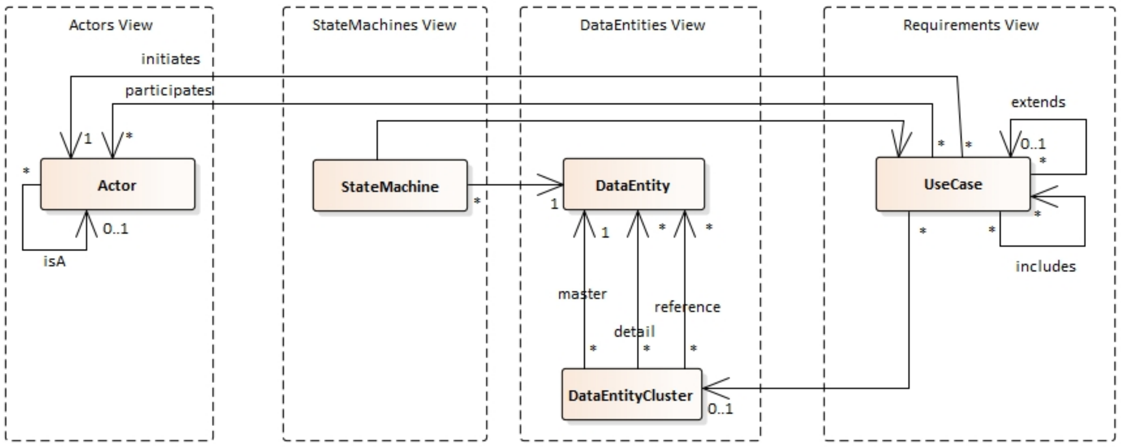

Fig.1. RSL partial metamodel with constructs organized in views.

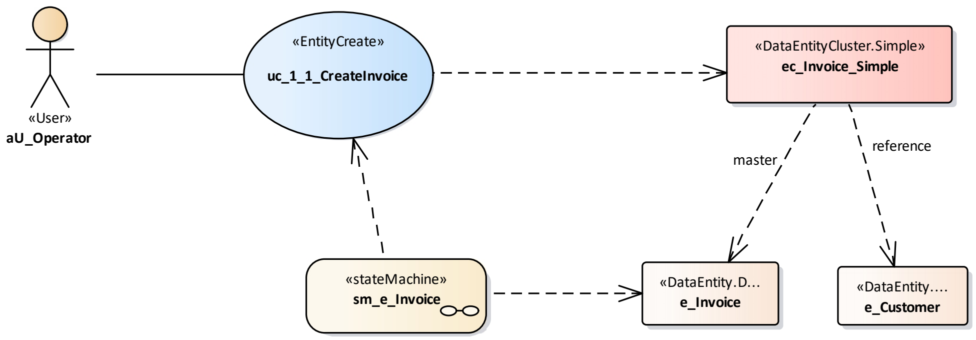

Fig.2.UML-like diagram suggesting the relationships between RSL's elements.

2.1 DataEntities and DataEntityClusters

DataEntities represent *the structural entities that exist in an information system,* commonly associated to data concepts captured and identified from the domain analysis. These entities can be later on represented by data structures like tables or collections of some system's database. As suggested in Figure 1 this view includes the definition of two key constructs: DataEntities and DataEntityClusters, which grammars are expressed in Specification 1 and 2.

A ***DataEntity*** 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*. *DataEntity type* values are "Parameter", "Reference", "Master", "Document" and "Transaction" based on their functions and the type of data that they serve [15]. *DataEntity subtype* values are "Master" and "Detail" meaning, respectively, that the data entity has its own relevance and identification (e.g., the Invoice entity) or its existence depends on other entity (e.g., the Invoice-Line entity that depends on the Invoice entity). A DataEntity can also be defined as read-only (with the *isReadOnly* property) and encrypted (with the *isEncrypted* property). An ***Attribute*** denotes *a particular structural property of the respective DataEntity*. An attribute has an id, name, type (e.g., Integer, Double, String, Date, DateTime, Text) 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), and several constraints defined with the *isNotNull*, *isUnique*, *isDerived*, *isReadOnly* and *isEncrypted* properties.

```
DataEntity:
   'DataEntity' name=ID  (nameAlias=STRING)? ':' type=DataEntityType (':' subType=DataEntitySubType)? ('['
        ('isA' super=[DataEntity | QualifiedName] )?
        (isReadOnly='isReadOnly')?  (isEncrypted='isEncrypted')?
        (attributes+=DataAttribute)+
        (primaryKey=PrimaryKey)?
        (foreignKeys+=ForeignKey)*
        (checks+=Check)* ']')?;

DataAttribute:
   'attribute' name=ID  (nameAlias=STRING)?  ':' type=DataType ('['
        ('multiplicity' multiplicity=Multiplicity)?
        ('defaultValue' defaultValue=STRING)?
        ('values' values=STRING)?
        (isNotNull='isNotNull')? (isUnique='isUnique')?   (isDerived='isDerived')?
        (isReadOnly='isReadOnly')?  (isEncrypted='isEncrypted')?  ']')?;
```

Spec. 1. Definition of DataEntity and respective Attributes.

On the other hand, a **DataEntityCluster** construct denotes a *cluster of several structural entities* (i.e., DataEntities) that present logical arrangements among themselves. Each data entity shall have a specific role in that cluster, namely: “master” role identifies the main DataEntity (e.g. Invoice); “detail” role identifies a partOf or child DataEntity (e.g., InvoiceLine); and “reference” role identifies a dependent or usedBy DataEntity (e.g., Customer). A DataEntityCluster is a construct particularly relevant to associate to UseCases (as suggested in Figures 1 and 2). A DataEntityCluster shall refer at least one DataEntity with the “master” role, and optionally other DataEntities with “detail” and/or “reference” roles. A DataEntityCluster is classified in a type with three values according the respective level of complexity: (1) VerySimple, if it refers only one DataEntity with the master role; (2) Simple, if it refers a master but at least one reference DataEntity; and (3) Complex, if it refers a master and at least one detail DataEntity.

```
DataEntityCluster:
   'DataEntityCluster' name=ID  (nameAlias=STRING)?  ':' type=DataEntityClusterType ('['
        (master=MasterDEntity)
        (details+=DetailDEntity)*
        (references+=ReferenceDEntity)* ']')?;
```

Spec. 2. Definition of DataEntityCluster.

2.2 Actors

Spec. 3 shows the grammar for the Actor rule definition. Actors represent *the participants of use cases* or *user stories*. Actors represent *end-users* and *external systems* that interact directly with the system under study, and in some particular situations can represent *timers* and other *complex conditions* that trigger the start of some use cases. Actors may depend only of Stakeholders, if relevant; that means that if stakeholders have been defined and if some of them were categorized as “User_Direct” or “System_External” they can also be defined as actors.

```
Actor:
   'Actor' name=ID  (nameAlias=STRING)?  ':' type=ActorType ('['
        ('isA' super=[Actor | QualifiedName] )?
        ('stakeholder' stakeholder=[Stakeholder | QualifiedName] )? ']')?;
```

Spec. 3. Definition of Actors.

2.3 UseCases

Spec. 4 shows the partial grammar of the UseCase rule definition. Traditionally a use case means a sequence of actions that one or more actors perform in a system to obtain a particular result. However, the RSL's UseCase construct extends this general and vague definition considering some additional aspects, namely: (1) A use case shall be classified by an extensive set of use case types corresponding to a well-defined use case patterns commonly found in information systems, such as EntitiesManage, EntityCreate, EntityUpdate, and many others; (2) A use case shall be applicable to a data entity or a cluster of data entities (DataEntityCluster); (3) 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 or complex conditions; (4) A use case can define pre and post conditions; (5) A use case can define several actions that may occur in its context, some of them very common in information systems such as user actions (e.g., Search, Filter, Create, Read, Update, Delete, Print, Close, Cancel) and other customized actions (e.g., in the running example, the actions ConfirmPayment, Approve or Reject invoices); (6) A use case can define "includes" relations to other use cases; this relation means that the behavior of the included use case is added to the source use case in a given point of its execution; (7) A use case can define several extensions points available in its context; this means that it can be extended by other uses cases in these specific points and based on some "extends" relations; (8) A use case can extends the behavior of other use case (the target use case) in its specific extension point; (9) 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 several alternative and exception scenarios; (10) A use case scenario is defined by a set of sequential or parallel steps; (11) A use case step is classified by a set of types and defined as simple or complex step; steps are triggered by the actor or by the system, respectively with steps of type (actor) prepares data or calls the system, or (system) executes an action or returns data.

```
UseCase:
    'UseCase' name=ID  (nameAlias=STRING)? ':' type=UseCaseType ('['
          ((isNegative ?= 'isNegative') | (isPositive ?= 'isPositive'))?
          ((isConcrete ?= 'isConcrete') | (isAbstract ?= 'isAbstract'))?
          ((isSolution ?= 'isSolution') | (isProblem ?= 'isProblem'))?
          ('stakeholder' stakeholder=[Stakeholder | QualifiedName] )?
          ('actorInitiates' actorInitiates=[Actor | QualifiedName] )
          ('actorParticipates' actorParticipates+=RefActor)?
          ('dataEntity' dataEntity=[DataEntityGeneric | QualifiedName] )?

          (actions= UCActions)?
          (extensionPoints= UCExtensionPoints)?
          (includes= UCIncludes)?
          (extends+= UCExtends)*

          scenarios+=Scenario*  ']')?;
```

Spec. 4. Definition of UseCase.

RSL offers a rigorous way to specify use cases, in particular when compared with UML or SysML. Its users can adopt different styles in what concerns use cases specification, depending on their needs and preferences. For example, in an initial phase of a project or in an agile project, the users might prefer to just specify use cases without many details (such as without defining scenarios and steps). On the other hand, if users require an extensive level of specification, their specifications can include the description of scenarios and steps in a detailed way.

2.4 StateMachines

Spec. 5 shows the partial grammar of the StateMachine rule definition. StateMachines define the behaviour of DataEntities in their relationships with use cases. A StateMachine is necessarily assigned to one DataEntity or DataEntityCluster (i.e., a DataEntityGeneric) and classified as *Simple* or *Complex* depending on the number of involved states and transitions (e.g., a StateMachine with more than three 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., states like Created, Pending, Approved, Rejected). In addition, a state can be marked 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 case's 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:
   'StateMachine' name=ID  (nameAlias=STRING)?  ':' type=StateMachineType ('['
        'dataEntity' entity= [DataEntityGeneric | QualifiedName]
        states= States ']')?;

States:  {State} states += State*;
State:
   'state'name=ID  (nameAlias=STRING)?
        (isInitial ?= 'isInitial')? (isFinal ?= 'isFinal')?
        ('onEntry' onEntry= STRING)?  ('onExit' onExit= STRING)?
        (':' (transitions+= Transition))? (transitions+= Transition)* ;

Transition: (ucAction= RefUCAction ('nextState' nextstate= [State])?);

RefUCAction: 'useCase' useCase=[UseCase | QualifiedName]  'action' action= [UCAction | QualifiedName];
```

Spec. 5. Definition of DataEntity's StateMachines.

3. An Illustrative Example

To support the discussion of the RSL language we introduce a fictitious information system called *"Invoice Management Ssystem*" (IMS) as it is found in many ERPs. The following text describes partially a variety of informal requirements (for the sake of simplicity we use the same example as followed in other papers that described different aspects of the RSL, e.g. as found in [25,27]):

IMS 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 user roles, such as operator, manager and administrator.

User-administrator shall be responsible for managing users, configuring technical features (e.g., user roles, export configuration parameters, general enterprise information). System shall allow user-administrator to register users. During this process the administrator shall specify first and last name of the user, email address, and username. User password shall be automatically generated by the system and sent to its user email.

User-operator is responsible for managing customers and invoices. User-operator shall create/update information related to customers and invoices. For each customer the system shall maintain the following information: name, fiscal id, logo image, address, bank information and additional information such as basic person contact information. System shall allow user-operator to define some customers as VIP. If a customer is defined as VIP, for each of her invoice, the system shall allow user-operator to set a predefined discount tax. That amount of discount can change throughout the time and depends on the current policy of the company. For each product the system shall maintain the following information: name, description, price, VAT category, and VAT value. Product must have only one VAT category and maintain the respective current VAT value.

User-operator shall create new invoices (with respective invoice details). However, 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.

The System shall automatically alert the user-manager, for all the invoices that were sent to customers but not yet paid, after 30 days of their respective issue date. In the beginning of each year the System shall archive and send to the ERP-System all paid invoices of the last year. [...]

3.1 DataEntities and DataEntityClusters

Consider the following snippet adapted from the IMS description:

For each ***customer*** *the system shall maintain the following information: name, fiscal id, logo image, address, bank information and additional information such as basic person contact information. System shall allow user-operator to define some customers as VIP. If a* ***customer is defined as VIP****, for each of her invoice, the system shall allow user-operator to set a predefined discount tax. For each* ***product*** *the system shall maintain the following information: name, description, price, VAT category, and VAT value [...] User-operator shall create new* ***invoices*** *(with respective* ***invoice details****) [...] In addition, for each* ***invoice****, the user-operator needs to keep track if it is paid or not [...]*

Some text fragments are identified and annotated, namely: (i) the **data entities** (bold text), e.g., invoice, customer, product; and the data entity attributes (text marked with light blue (light gray)), e.g., name, fiscal id. Spec. 6 shows a partial rigorous specification of these data entities in RSL, and Figure 3 illustrates an equivalent UML-like representation. In addition, on top of these data entities, Spec. 7 shows the specification of DataEntityClusters.

```
DataEntity e_VAT "VAT Category" : Reference […]
DataEntity e_Product "Product" : Master […]
DataEntity e_Customer "Customer" : Master […]
DataEntity e_CustomerVIP "CustomerVIP" : Master […]

DataEntity e_Invoice "Invoice" : Document [
  attribute ID "Invoice ID" : Integer [isNotNull isUnique]
  attribute customerID "Customer ID" : Integer [isNotNull]
  attribute dateCreation "Creation Date" : Date [defaultValue "today" isNotNull]
  attribute dateApproval "Approval Date" : Date
  attribute datePaid "Payment Date" : Date
  attribute dateDeleted "Delete Date" : Date
  attribute isApproved "Is Approved" : Boolean [defaultValue "False"]
  attribute totalValueWithoutVAT "Total Value Without VAT" : Decimal(16.2) [isNotNull]
  attribute totalValueWithVAT "Total Value With VAT" : Decimal(16.2) [isNotNull ]
  foreignKey e_Customer(customerID)]

DataEntity e_InvoiceLine "InvoiceLine" : Document: Detail [
  attribute ID "InvoiceLine ID" : Integer [isNotNull isUnique]
  attribute invoiceID "Invoice ID" : Integer [isNotNull]
  attribute order "InvoiceLine Order" : Integer [isNotNull]
  attribute productID "Product ID" : Integer [isNotNull]
  attribute valueWithoutVAT "Value Without VAT" : Decimal
  attribute valueWithVAT "Value With VAT" : Decimal
  primaryKey (ID)
  foreignKey e_Invoice(invoiceID)
  foreignKey e_Product(productID)
  check ck_InvoiceLine1 "isUnique(invoiceID+order)" ]
```

Spec. 6. Example of a partial RSL specification of DataEntities.

```
DataEntityCluster ec_Customer "Customers" : Simple [master e_Customer]
DataEntityCluster ec_Product "Products" : Simple [master e_Product reference e_VAT]
DataEntityCluster ec_Invoice "Invoices (Complex)" : Complex  [
  master e_Invoice
  detail e_InvoiceLine [reference e_Product, e_VAT]
  reference e_Customer]
```

Spec. 7. Example of a partial RSL specification of DataEntityClusters.

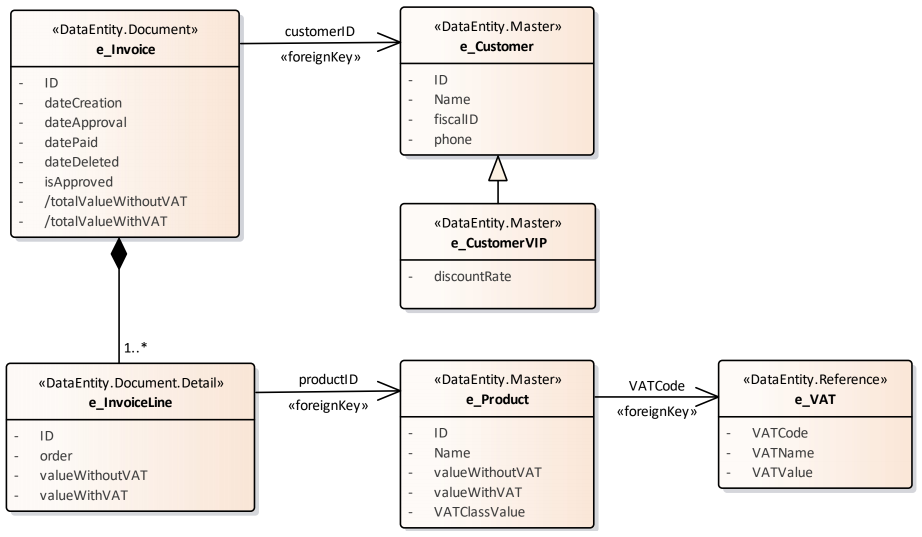

Fig. 3. UML-like diagram representing DataEntities.

3.2 Actors

Actors are particularly identified and annotated as dashed underlined text from the IMS description:

User-administrator shall be responsible for […] User-operator is responsible for […] Before sending an invoice to a customer, the invoice shall be formally approved by the user-manager. The System shall […] after 30 days of their respective issue date. In the beginning of each year the System […] sends to the ERP-System […]

From this analysis we systematically specify the actors of the IMS as suggested in Spec. 8.

```
Actor aU_TechnicalAdmin "TechnicalAdmin" : User [description "Admin manage Users, VAT, etc."]
Actor aU_Operator "Operator" : User [description "Operator manages Invoices and Customers"]
Actor aU_Manager "Manager" : User [description "Manager approves Invoices, etc."]
Actor aU_Customer "Customer" : User [description "Customer receives Invoices to pay"]
Actor aS_ERP "ERP" : ExternalSystem [description "ERP receives info of paid invoices"]
Actor aT_BeginningOfYear : Timer [description "Beginning of each Year"]
Actor aT_InvoiceNotPaidAfter30d : Timer [description "Invoices not paid after 30d of issue"]
```

Spec. 8. Example of a partial RSL specification of Actors.

3.3 Use Cases

Consider the following snippet adapted from the IMS description:

*User-operator shall be responsible for managing **users**, configuring **technical features** [...]*

*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 approved or rejected 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 the **invoice** is paid or not.*

*The System shall automatically trigger an alert for all the **invoices** not yet paid, after 30 days of their respective issue date. The System shall archive and send to the ERP-System all paid **invoices** whose creation date is older than 1 year. User-manager shall allow monitoring the process of creating, approving and payments **invoices**. User-manager shall approve or reject **invoices** [...]*

Some textual fragments are particularly identified and annotated, namely: (i) the data entities (with **bold** text), e.g., invoices, customers, products; (ii) the use cases (with underlined text), e.g., manage invoices, manage users, issue and send invoice; and (iii) the actors involved in such use cases (with dashed underlined text), e.g., user-operator, user-manager, customer, ERP-System.

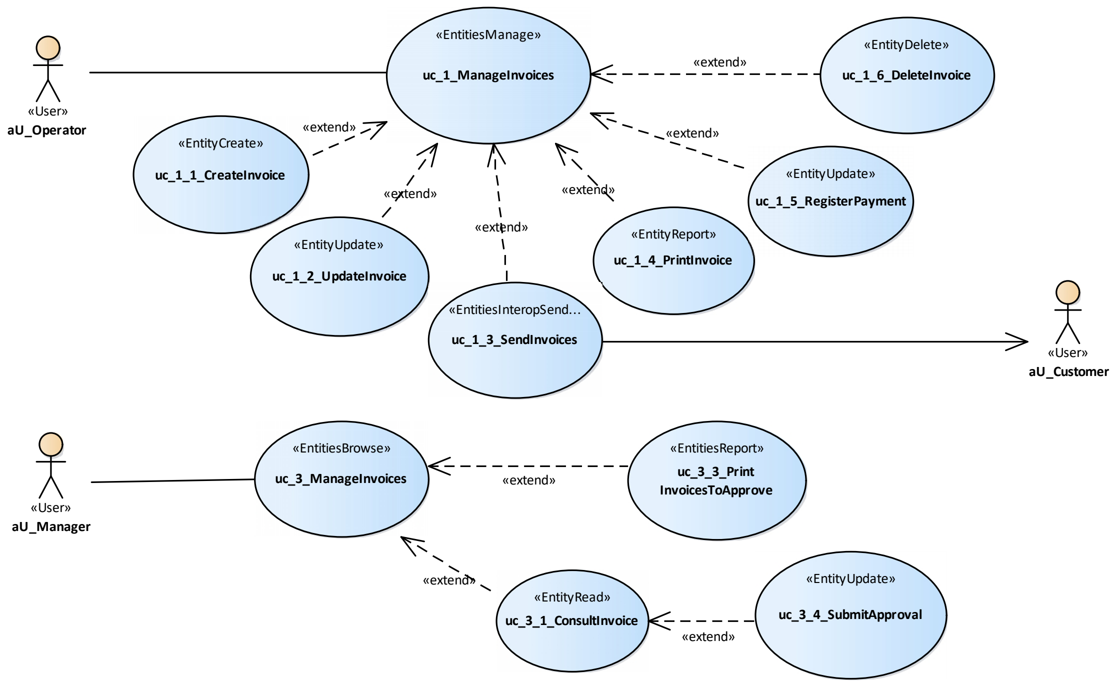

Fig.4.UML-like diagram representing UseCases.

```
UseCase uc_1_ManageInvoices "Manage Invoices" : EntitiesManage […]
UseCase uc_1_1_CreateInvoice "Create Invoice" : EntityCreate […]
UseCase uc_1_2_UpdateInvoice "Update Invoice" : EntityUpdate […]

UseCase uc_1_3_SendInvoices "Send Invoices" : EntitiesInteropSendMessage [
       actorInitiates aU_Operator  actorParticipates aU_Customer
       dataEntity ec_Invoice
       actions aSend, aClose
       extends uc_1_ManageInvoices onExtensionPoint EPSendInvoices]
UseCase uc_1_4_PrintInvoice "Print Invoice" : EntityReport […]
UseCase uc_1_5_RegisterPayment "Register Payment" : EntityUpdate […]

UseCase uc_3_ManageInvoices "Manage Invoices (for Manager)" : EntitiesBrowse […]
UseCase uc_3_1_ConsultInvoice "Consult Invoice" : EntityRead [
       actorInitiates aU_Manager
       dataEntity ec_Invoice
       actions aFirst, aLast, aPrevious, aNext, aClose
       extensionPoints SubmitApproval
       extends uc_3_ManageInvoices onExtensionPoint Consult_Invoice]
UseCase uc_3_3_PrintInvoicesToApprove "Print Invoices To Approve" : EntitiesReport […]
UseCase uc_3_4_SubmitApproval "Submit Invoice Approval" : EntityUpdate […]
```

Spec. 9. Example of a partial RSL specification of UseCases.

From this text analysis we identify and then specify the data entities, actors and use cases systematically. Spec. 9 shows a partial specification of these use cases in RSL, and Figure 4 illustrates the equivalent UML-like representation. Due to space constraints Spec. 9 only shows the expanded specification of the use cases uc_1_3_SendInvoices, uc_3_1_ConsultInvoice; the others are collapsed or just omitted. The use case uc_1_3_SendInvoices is typed as EntitiesInteropSendMessage, is started by the actor aU_Operator and also participated by the aU_Customer that is the target of the send message..

3.4 StateMachine

Finally, Spec. 10 shows the specification of the e_Invoice's behaviour through the definition of its respective statemachine sm_e_Invoice, and Figure 5 depicts the equivalent UML representation for this same example. This statemachine includes six states: Initial, Pending, Approved, Rejected, Paid and Deleted. Initial is an initial state while Paid and Deleted are final states. When a data entity is in some state several use case's actions might occur, some of them may imply a state transition as defined by the respective "nextState" attribute. For example, when the invoice is in the state Pending, if the Manager is in the scope of the uc_3_4_SubmitApproval and approves the invoice (i.e. triggers the action aApprove), then the invoice changes to the state Approved.

```
StateMachine sm_e_Invoice "StateMachine_Invoice" : Complex [
        dataEntity e_Invoice
        description "StateMachine of entity Invoice"
        state Initial isInitial onEntry "In creation"
                  useCase uc_1_1_CreateInvoice action aCreate nextState Pending
        state Pending onEntry "e.state= 'Pending'; e.isApproved= False"
                  useCase uc_4_NotifyInvoicesToApprove action aNotify
                  useCase uc_3_4_SubmitApproval action aApprove nextState Approved
                  useCase uc_3_4_SubmitApproval action aReject nextState Rejected
        state Approved onEntry "e.state= 'Approved'; e.isApproved= True"
                  useCase uc_1_3_SendInvoices action aSend
                  useCase uc_5_AlertInvoicesNotPaid action aNotify
                  useCase uc_1_5_RegisterPayment action aConfirmPayment nextState Paid
        state Rejected onEntry "e.state= 'Rejected'; e.isApproved= False"
                  useCase uc_1_2_UpdateInvoice action ReSubmit2Approval nextState Pending
        state Paid isFinal onEntry "e.state= 'Paid'"
        state Deleted isFinal onEntry "e.state= 'Deleted'"]
```

Spec. 10. Example of a partial RSL specification of a DataEntity' StateMachine.

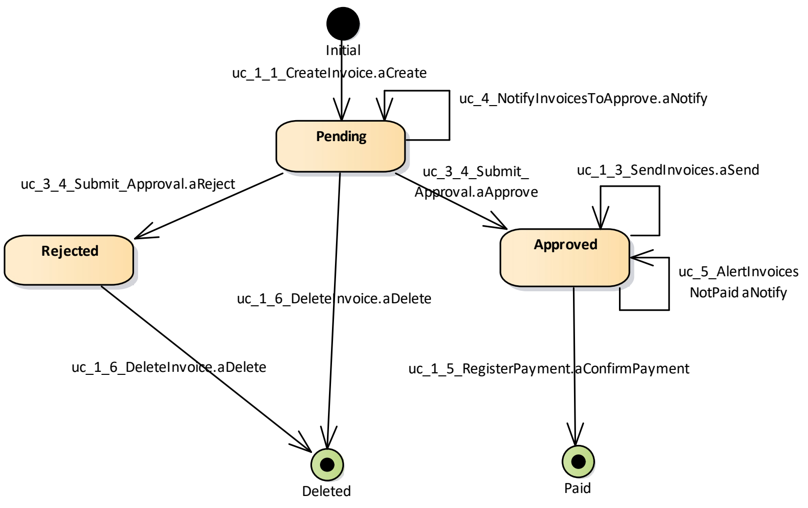
Initial
uc_1_1_CreateInvoice.aCreate
uc_4_NotifyInvoicesToApprove.aNotify
Pending
uc_3_4_Submit_Approval.aReject
uc_3_4_Submit_
Approval.aApprove
uc_1_3_SendInvoices.aSend
Rejected
Approved
uc_5_AlertInvoices
NotPaid aNotify
uc_1_6_DeleteInvoice.aDelete
uc_1_6_DeleteInvoice.aDelete
uc_1_5_RegisterPayment.aConfirmPayment
Deleted
Paid

Fig.5.UML-like diagram representing a DataEntity's StateMachine.

4. Related Work

The documentation of system requirements has consisted in creating descriptions of the application domain, as well as a prescription of what the system should do and other organizational, legal or technological constraints [16]. In general, these requirements are specified in natural languages due to their higher expressiveness and ease of use. However, the usage of these languages also presents drawbacks like ambiguity, inconsistency and incompleteness [10,11,16,18]. To reduce these problems, natural language specifications are many times complemented by some sort of other specifications and models that use controlled, formal or semi-formal modelling languages. These languages provide a set of constructs (e.g., actor, use case or user stories) that explicitly or implicitly define its abstract syntax and semantics, and that address different concerns and abstraction levels. In addition, these languages provide different notations or concrete syntaxes, such as textual, graphical, tabular, form-based representations [14,26].

Requirements specifications based on ***natural language*** are very expressive, easy to be written and read by humans, but not very precise because are ambiguous and inconsistent by nature, and hard to be automatically manipulated by computers [9]. Apparently the usage of ***formal language methods (like B or Z)*** could overcome these problems [21]. However, this would only address part of the problem, as we still need to take care while interpreting the natural language requirements to create a formal specification, given that engineers often misinterpret natural language specifications during the design phase. The same occurs with the attempt to directly create formal requirements specifications, especially when the real requirements are not discovered and validated at first by the business stakeholders. Thus, the usage of formal languages involves an additional misinterpretation level due to the typically complex syntax and mathematical background required [8]. Given that formal methods are expensive to apply creating formal requirements specifications might have a negative impact because they require specialized training and are time-consuming [28]. In the attempt of getting the best from both worlds – the familiarity of natural language and the rigorousness of formal language –, some approaches have proposed *controlled natural languages*, which are engineered to resemble natural languages [12]. However, they are only able to play the role of natural language to certain extend: while they are easy to read, they are hard to write without specialized tools [8]. On the other hand, many of these languages are restricted sets of English and targeted to support general technical documentation, like ACE [9], not so focused on RE like RSL. A ***modeling language*** **like** ***UML*** (and ***SysML*** at some extent) is a reasonable foundation for requirements modeling, but it is incomplete for modeling requirements because it lacks models that tie requirements to business value and models that present the system from an end user's point of view. In addition, its technical roots make it simply too complex for business stakeholders to adopt because its models are geared towards modeling the structure of the software architecture. UML is intended to be used to describe the technical design of a system and is at best retrofitted to model requirements. Differently from RSL, there are a number of drawbacks associated to the use of UML regarding this purpose. First, there is not a well-defined way to establish relationships between constructs defined for different types of diagrams, e.g., between use cases and classes, or between state machines and use cases: in RSL it is possible to establish such relationships in a rigorous way, respectively, between use cases and data entities, or between state transitions and use case's actions. Second, in UML there is not a standard way to further classify its constructs, e.g., use cases or actors do not have any further semantics: in RSL every construct has a type which allows enriching its meaning, e.g., use case types correspond to use case patterns commonly found in information systems. Third, UML use cases are defined in a high-level way with only stating the name and the relationships with other use cases and actors. On the other hand, RSL use cases provide a flexible and precise way to define this type of requirements: if intended, RSL use cases can be defined as simpler as UML, but in general, it is possible to add more information in a precise way, namely to express the list of actions that may be triggered, the extensions points that may be extended by other use cases, or the scenarios with further detail. ***RML*** [3] is one of the largest RE-specific language that gather many constructs and types of models, e.g., business data diagram or state diagram; data dictionary, system flow, state diagram. On the contrary to UML or SysML languages, RML also provides other representations such as tabular (e.g., roles and permissions matrix, state table), and form-based (e.g., use cases). However, RML is defined implicitly and consequently its models/specifications are expressive and flexible but less rigorous and precise in comparison with UML, SysML, SilabREQ, or RSL. Still in comparison with RSL, RML includes additional low-level constructs and models (e.g., permissions, user-interface flows, reports, display-action-response) but lacks others more specific to RE, like goals, constraints, or quality requirements. Furthermore, RSL is defined with a precise grammar with which its specifications are textually but consistently represented. **Modeling languages (like UML and SysML) or goal-oriented languages (like i*, KAOS or Tropos)** are less expressive than natural language and cannot be regarded as a common language to communicate requirements, because business stakeholders still require some training to understand them. Also, despite being easier to be understandable than formal languages, these modeling languages are regarded as less powerful in terms of analytical capabilities because they often lack tool support to enforce the implicit semantics of their modeling elements, or might even intentionally leave some unspecified parts of the language itself, in which case they are considered as semi-formal. Some authors even argue that the simplicity of these languages comes precisely from this lack of semantic enforcement: it is easy to create models because "anything goes". Furthermore, the usage of graphical languages cause another limitation when users include too much detail in the model, cluttering it and thus affecting its readability; or when the size of the models (i.e., the number of model elements involved) increases too much. Therefore, despite the existence of such graphical approaches, textual specifications are still regarded by many as the most suitable, fast, and preferred manner to initiate the requirements development process.

5. Conclusion

A requirements specification describes multiple concerns of a system and supports several stakeholders. For an effective communication every stakeholder shall communicate in a common natural language. Although natural languages are the preferred form for requirements representation, they exhibit some characteristics that contribute for several requirements quality problems. This paper describes and discusses the RSL language, which intends to mitigate some of these problems and also to improve the overall productivity of the RE processes. RSL includes a large set of constructs logically arranged into views according to the most common RE concerns. Each construct can be defined as a linguistic pattern and represented by a well-defined RSL's linguistic style. In spite of this, these constructs can also be represented by other linguistic styles defined in a project or organization basis accordingly the needs of its users. These constructs can also be represented graphically such as with UML, RML or UML-based profiles as we informally showed in this paper. Furthermore, these constructs can have other representations, like tabular or form-based.

RSL supports requirement engineers to produce requirements specifications at different levels of detail, with different concerns and different types of requirements, specifically: goals, functional requirements, quality requirements, constraints, user stories, and use cases. In addition, RSL provides other constructs (e.g., terms, stakeholders, actors, data entities, or state machines) that, in spite of not being classified as "requirements" in RSL, are still important to complement and enrich such specifications. This paper focuses the discussion just on the RSL's use case constructs and, it shows that it is possible to rigorously specify use cases with actors, data entities and state machines, and to establish proper traceability between these constructs, for example as suggested in Figure 1. In spite of other proposals for RE-specific templates or linguistic patterns, and as far as we know from the literature review, RSL is the first that provides such a comprehensive language that integrates a large number of constructs that can be represented in a consistent and systematic way.

For future work, we plan to disseminate the adoption and practice of RSL in both academia and industry, to conduct real world case studies, and to research model-driven techniques [26,29] to automatically validate and transform RSL specifications into technical documentation and visual models.

Acknowledgements

Work supported by funds under FCT UID/CEC/50021/2019 and 02/SAICT/2017/29360.

References

1. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Pub. (2016)
2. Caramujo, J., et al.: RSL-IL4Privacy: A Domain-Specific Language for the Specification of Privacy-Aware Requirements, in Requirements Engineering, Springer, 24(1) (2019)

3. Chen, A., Beatty, J.: Visual models for software requirements. Microsoft Press (2012)
4. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2001)
5. Eveleens, L., Verhoef, C.: The Rise and Fall of the Chaos Report Figures, IEEE Software (2010)
6. Fernandes, J.M., Machado, R. J.: Requirements in engineering projects, Springer (2016)
7. Ferreira, D., Silva, A. R.: RSL-IL: An Interlingua for Formally Documenting Requirements. In Proc. of the of 3rd IEEE Workshop on Model Driven Requirements Engineering, IEEE CS (2013)
8. Foster, H., et al.: Assertion-based Design. Springer (2004)
9. Fuchs, N., Kaljurand, K., Kuhn, T.: Attempto controlled english for knowledge representation. In Reasoning Web, 104-124, Springer (2008)
10. IEEE: IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications (1998)
11. Kovitz, B.: Practical Software Requirements: Manual of Content and Style. Manning (1998)
12. Kuhn, T.: A survey and classification of controlled natural languages. Computational Linguistics, 40(1), 121-170 (2014)
13. Maciel, D., Paiva, A. C. R., Silva, A. R.: From Requirements to Automated Acceptance Tests of Interactive Apps: An Integrated Model-based Testing Approach. In Proc. of ENASE'2019, SCITEPRESS (2019).
14. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM Computing Surveys, 37(4):316-344, (2005)
15. Microsoft, Data entities and Categories of entities, Dynamics 365 White paper, 2017. Accessed from https://docs.microsoft.com/en-us/dynamics365/unified-operations/dev-itpro/data-entities/data-entities?toc=/fin-and-ops/toc.json#categories-of-entities
16. Pohl, K.: Requirements Engineering: Fundamentals, Principles, and Techniques. Springer (2010)
17. Ribeiro, A., Silva, A. R.: Evaluation of XIS-Mobile, a Domain Specific Language for Mobile Application Development, Journal of Software Engineering and Applications, SCIRP, 7(11), 906-919 (2014)
18. Robertson, S. and Robertson, J.: Mastering the Requirements Process, Addison-Wesley (2006)
19. Savic, D., et al.: SilabMDD: A Use Case Model Driven Approach. In Proc. of 5th International Conference on Information Society and Technology (2015)
20. Savic, D., et al: SilabMDD: A Use Case Model Driven Approach. In Proc. of ICIST 2015 (2015)
21. Schneider, S.: The B-method: an introduction. Palgrave Macmillan (2001)
22. Silva, A. R., et al.: Improving the Specification and Analysis of Privacy Policies: The RSLingo4Privacy Approach. In Proc. of International Conference on Enterprise Information Systems, SCITEPRESS (2016)
23. Silva, A. R., Saraiva, J., Ferreira, D., Silva, R., Videira, C.: Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools. IET Software, IET, 1(6) (2007)
24. Silva, A. R., Saraiva, J., Silva, R., Martins, C.: XIS – UML Profile for eXtreme Modeling Interactive Systems. In Proc. of MOMPES'2007, IEEE Computer Society (2007)
25. Silva, A. R.: Linguistic Patterns and Linguistic Styles for Requirements Specification (I): An Application Case with the Rigorous RSL/Business-Level Language. In Proc. of EuroPLOP, ACM (2017)
26. Silva, A. R.: Model-Driven Engineering: A Survey Supported by a Unified Conceptual Model. Computer Languages, Systems & Structures, Elsevier, 43 (C), 139–155 (2015)
27. Silva, A.R., Paiva, A.C.R., Silva V.E.R.: A Test Specification Language for Information Systems Based on Data Entities, Use Cases and State Machines. In Model-Driven Engineering and Software Development, Communications in Computer and Information Science, Springer, 991 (2019)
28. Sommerville, I. and Sawyer, P.: Requirements Engineering: A Good Practice Guide. Wiley (1997)
29. Stahl, T., Volter, M.: Model-Driven Software Development. Wiley (2005)
30. Standish Group: Chaos Summary 2009 Report, The 10 Laws of Caos (2009)
31. Verelst, J., et al.: Identifying Combinatorial Effects in Requirements Engineering. In Proceedings of Third Enterprise Engineering Working Conference (EEWC 2013), Advances in Enterprise Engineering, LNBIP, Springer (2013)
32. Videira, C., Silva, A. R.: Patterns and metamodel for a natural-language-based requirements specification language. In Proceedings of CAiSE, (2005)
33. Withall, S.: Software Requirements Patterns. Microsoft Press (2007)