IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 16, 2021, accepted October 5, 2021, date of publication October 14, 2021, date of current version October 27, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3120004

Linguistic Patterns, Styles, and Guidelines for Writing Requirements Specifications: Focus on Use Cases and Scenarios

ALBERTO RODRIGUES DA SILVA
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1649-004 Lisboa, Portugal
e-mail: alberto.silva@tecnico.ulisboa.pt

This work was supported in part by the Portuguese Fundação para a Ciência e a Tecnologia (FCT) under Grant UIDB/50021/2020 and Grant 02/SAICT/2017/29360.

ABSTRACT A system requirements specification is a technical document extensively used during the respective system life cycle. It gathers the concerns and needs of various stakeholders, describes the common vision of that system, and therefore supports its development and operation processes. The popular form to write requirements specifications is with natural languages that, however, exhibit characteristics that often introduce quality problems, such as inconsistency, incompleteness, and vagueness, which shall be mitigated or avoided to some extent. This paper is part of a series of papers that have discussed linguistic patterns and linguistic styles to produce technical documentation more systematically and consistently. Specifically, this paper proposes a cohesive set of patterns and styles to better write use cases and scenarios. It also presents 38 practical guidelines and supports the discussion with concrete pedagogical examples represented with different styles, namely: visual use cases diagram (UML), a rigorous requirements specification language (RSL), and two informal controlled natural languages, one with a compact (CNL-A) and another with a more complete and verbose writing style (CNL-B). We conducted a pilot evaluation session with 24 subjects who provided encouraging feedback, with positive scores in all the analyzed dimensions. Based on this feedback, we may conclude that the adoption of these patterns, styles, and guidelines would help to produce better requirements specifications, written more systematically and consistently.

INDEX TERMS Requirements specification, linguistic pattern, linguistic style, controlled natural language, use case, structured scenario.

I. INTRODUCTION

Requirements engineering (RE) is a discipline that defines a common vision and understanding of socio-technical systems among the involved stakeholders and throughout their life cycle [1], [2]. The negative consequences of ignoring early RE activities are extensively reported and discussed in the literature [3], [4].

A system requirements specification (or just ''requirements specification'') is a technical document that defines and organizes the concerns of such systems from the RE perspective. A good requirements specification offers several benefits as reported in the literature such as [2], [5], [6]: contributes to the establishment of an agreement and business contract between customers and suppliers; provides a common ground for supporting the project budget and schedule estimation and plan; supports the project scope validation and verification, and may support deployment and future maintenance activities. It is usually recommended that a requirements specification shall be defined accordingly as a predefined template as well as a set of recommendations on how to customize and use it. In general, these templates recommend the use of various views and constructs (e.g., actors, use cases, user stories) that might be considered ''modular artifacts'' in the sense of their definition and reuse. Because there are dependencies among these constructs, some authors have argued that it is essential to minimize or prevent them, and some of these templates give support in this respect [7], [8].

Because requirements specifications support both technical and business stakeholders, they are usually written in natural languages. Indeed, natural languages reach an adequate communication level since they are flexible, universal, and humans are proficient at using them to communicate. So, they

The associate editor coordinating the review of this manuscript and approving it for publication was Laurence T. Yang.

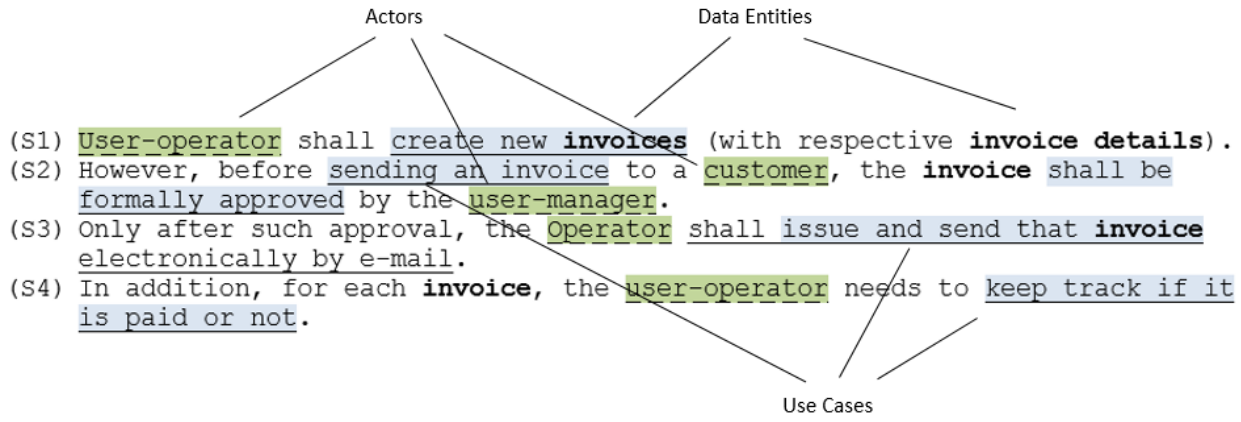

FIGURE 1. Example of informal requirements that involve use cases and related elements.

have minimal adoption resistance as a requirements documentation technique. Nevertheless, they also exhibit intrinsic features that frequently put them as the source of quality problems like inconsistency, incompleteness, incorrectness, and ambiguousness [1], [9]–[11].

For instance, a use case is a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal [12]. Despite being a popular manner to organize and define (mostly functional) requirements, there are still difficulties in both modeling and write use cases. As suggested in Fig. 1, use cases and related elements, like actors and data entities, are usually referred to in multiple requirements scattered all over the text and, unfortunately, inconsistently represented with different writing styles. For example, in Fig. 1, the names ''User-operator'' (in S1), ''Operator'' (in S3), and ''user-operator'' (in S4) should refer to the same actor or user role; or the sentences ''sending an invoice to a customer'' (in S2) and ''issue and send that invoice'' (in S3) may refer the same use case. These are simple but representative examples of inconsistency and ambiguity inherent to the natural language specification.

Due to these problems, some authors have proposed *practical recommendations for writing requirements effectively*, including general guidelines like the followings [1], [11], [13], [14]: the language should be simple, clear, and precise; should follow a standardized format to give coherence and uniformity to all sentences; the sentences should be simple, short, and written in an affirmative and active voice style; or that the vocabulary should be limited.

Furthermore, concerning the writing of use cases and scenarios, some authors have proposed more specific guidelines such as [15]–[19]. For instance, Lilly discusses several ''pragmatic cures'' for the writing of use cases, such as [18]: Be explicit about the scope; Draw the system boundary (at least in your head); Name the use cases from the perspective of the Actor's goals; Get agreement early in the project about the use of actor names (and other terms); Make sure that the granularity of the use cases is appropriate; The actors may be defined too broadly, or The granularity of the use case may be too coarse. Constantine and Lockwood discuss and recommend avoiding the specification of concrete use cases, i.e., those that refer to concrete terms (i.e., ''John'', ''Mary'') or specific user interfaces (e.g., ''click in the Ok button'', ''customer panel'') and replace these by abstract, generalized, and technology-free descriptions of the essence of a problem, named as ''essential use cases'' [17]. Cockburn discusses several types of use cases (e.g., casual vs dressed, business vs. system, white-box vs. black-box) and discusses his preferred format for the writing of use case scenarios based on the following guidelines [15]: one column of plain prose; numbered steps; no ''if'' statements; use the numbering convention in the extensions sections involving combinations of digits and letters. Wirfs-Brock and Schwartz also discusses several pragmatic recommendations for writing use cases and introduces the conversational (two-columns) form of use cases [16], then popularized by Constantine and Lockwood [17].

However, some of these recommendations are still general or difficult to apply in practice and need to be further elaborated and exemplified considering specific constructs as supported by *controlled natural languages*. In this scope, we introduce *linguistic patterns* as grammatical rules that allow their users to write correctly in a common language. Despite the diversity of terms found in the literature – for instance, ''syntactic requirements pattern'' [1] or ''requirements template'' [20] –, we assume in this paper the terms ''*linguistic pattern*'' and ''*linguistic style*'', as discussed by Silva [21], to mean, respectively, the *definition* and the *representation* of such grammatical rules.

In his paper, Silva focused on business-level constructs, namely on constructs like glossary terms, stakeholders, business goals, processes, events, and flows [21], and recently on data entities [22]. Although inspired by that work, this paper is substantially different because it is focused on the *textual specification of use cases and use case scenarios*,

and discusses not only *patterns and writing styles* but also gathers and discusses the respective *pragmatics, i.e., practical recommendations and guidelines* for writing use cases and scenarios (some of them previously discussed by other authors). Besides, and on the contrary of other proposals, this paper *supports the discussion with a unique and rich running example* that includes common and pedagogical situations. This example refers to the requirements of a fictitious information system called ''BillingSystem'', which is an invoice management system, inspired and adapted from the example initially introduced by Silva [21], [22].

The patterns discussed in this paper allow requirements engineers and product designers to write systematically and rigorously use case's aspects, including actors, use cases, use case relationships, and use case scenarios. To better support the discussion, these linguistic patterns are represented according to three linguistic styles, namely by a rigorous requirements specification language (RSL) and two informal controlled natural languages, the CNL-A (with a compact representation) and the CNL-B (with a more verbose representation). Also, we add practical guidelines to help to better write such aspects. (Of course, the interested reader might adjust these styles or define a new one according to her preferences or needs.)

This paper is structured into seven sections. Section 2 introduces the relevance of controlled natural languages to writing requirement specifications and introduces the three languages used in this research. Section 3 introduces the core notions adopted in the scope of this research, i.e., the notions of linguistic pattern and linguistic style. Sections 2 and 3 are similar to those of the paper [22], however, they are included in this paper for self-completeness for those readers that did not read that previous paper. Section 4 discusses the proposed use cases' linguistic patterns and supports the debate with a running example. Section 5 refers to and discusses the related work. Section 6 presents the conditions and results of a pilot user session conducted to receive feedback from IT professionals and students. Lastly, Section 7 presents the conclusion and the open topics of research. Moreover, Appendix A describes the running example used in Section 4 and includes a draft representation of that example with the discussed writing styles as well as a UML use case diagram. Appendix B summarizes the linguistic patterns and the recommended vocabulary.

II. CONTROLLED NATURAL LANGUAGES

Requirements specifications are usually written with natural language and hence are expressive, easy to be read and written by humans, but not very precise because they tend to be ambiguous and inconsistent by nature and hard to be automatically manipulated by computers [23]. The usage of formal language methods could overcome some of these problems [24]. Although, that only addresses part of the question due to its difficulty in applying them into not-so-critical systems because they require specialized training and are time-consuming. In the attempt to get the best from both worlds, i.e., the familiarity of natural languages and the rigorousness of formal languages, some authors have proposed controlled natural languages, which are engineered to resemble natural languages [25].

A controlled natural language (CNL) defines a constrained use of a natural language's grammar (syntax) and a set of terms (comprising the semantics of these terms) to be used in the constrained grammar [1], [2]. The adoption of CNLs may have the following benefits: CNL sentences are easy to understand since they are like sentences in natural language; however, they are less ambiguous than expressions in natural language, since they have simplified grammar and a predefined vocabulary with precise semantics; and are semantically correct and can be computational manipulated since they may have a formal grammar and predefined terms.

For the sake of the explanation of this paper, we consider three different linguistic styles represented by two informal CNLs (named as CNL-A and CNL-B) and a rigorously defined CNL (the RSL language).

A. CNL-A AND CNL-B LANGUAGES

CNL-A and CNL-B are informally defined as follows.

CNL-A is intended to express statements in a *compact writing style*, namely according to the following template: ''<**id**> **is a** <**type**> <**element**> **[, extends** <**isA**>**]?.**'', in which ''<id>'', ''<type>'' ''<element>'' and ''<isA>'' are template fragments that are replaced by concrete text fragments. Examples of sentences with this language can be:

```
kUser is a Person Stakeholder.
Customer is a User Actor.
CustomerVIP is a User Actor, extends a Customer.
```

On the other hand, CNL-B intends to express statements in a *more verbose, expressive, and complete writing style*, based on the following template: ''< **element**><**id**> **[(**<**name**>**)]? is a** <**type**> **[and a** <**isA**>**]? [, described as** <**description**>**]?.**''. Some equivalent sentences written with this language can be the following:

```
Stakeholder KeyUser (Key User) is a Person, described as
  a user representative.
Actor Customer is a User, described as a user that buys
  products from the eShop.
Actor CustomerVIP (Customer VIP) is a User and a Customer,
  described as a user that buys products from the eShop
  with a special discount program.
```

B. RSL LANGUAGE

RSL is a controlled natural language designed for the rigorous specification of requirements and tests [21], [26], [27]. RSL includes several constructs logically classified according to two dimensions [26]: abstraction level and specific concerns. The abstraction levels are business, application, software, and hardware levels. The concerns are active structure (subjects), behavior (actions), passive structure (objects), requirements, tests, relations and sets, and other concerns. From a syntactical perspective, any construct can be used in any type of system regardless of its abstraction level.

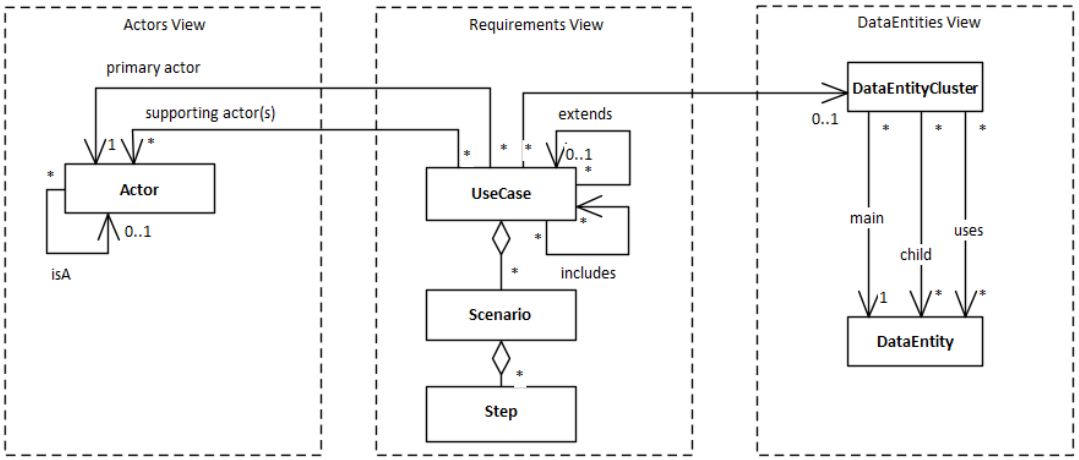

FIGURE 2. RSL partial metamodel with use cases and related constructs.

Fig. 2 shows the RSL partial metamodel that involves the constructs mostly related to this paper, particularly focused on use cases specification. For example, Fig. 2 suggests that: a use case may aggregate a set of scenarios; there are different types of relationships between use cases, namely extend and include relationships; a use case may use one data entity or a cluster of data entities; a use case shall have one primary actor (who has the use case's goal) and might be participated or supported by other actors.

The examples illustrated above can be represented with RSL as:

```
Stakeholder KeyUser ``Key User'': Person [description ``a
  user representative'']
Actor Customer: User [description ``a user that buys
  products from the eShop'']
Actor CustomerVIP ``Customer VIP'': User [isA Customer
  description ``a user that buys products from the eShop
  with a special discount program'']
```

III. LINGUISTIC PATTERNS AND LINGUISTIC STYLES

As discussed originally by Silva [21], a *linguistic pattern* is a set of rules that defines the elements and the vocabulary that shall be used in the sentences of these requirements technical documents. An *element rule* consists in a set of element attributes (e.g., <id>, <name> or <type>) defined by the following properties: name, type and multiplicity. On the other hand, a *vocabulary rule* defines a set of literal terms (e.g., ''User'', 'ExternalSystem'', 'Timer'') used to categorize some element attributes and to restrict the use of a limited number of terms. For example, the linguistic pattern Actor is defined by the following set of rules (i.e., the Actor element rule and the ActorType vocabulary rule):

```
Actor ::
  <id:ID> <name:String> <type:ActorType>
  <stakeholder:Stakeholder>? <isA:Actor>?
  <description:String>?

enum ActorType::
  User | ExternalSystem
```

The Actor element rule defines its element attributes (e.g., <id>, <name>, <type>, <isA>, <stakeholder>, <description>) and for each attribute the respective name (e.g., name, type, isA), type (e.g., ID, String, ActorType, Stakeholder) and multiplicity (e.g., '?') properties. The multiplicity of an attribute is not represented by default (and in this case means "1", a mandatory attribute), or can be represented by the following characters '?', '+', and '∗' meaning, respectively, 0..1 (optional), 1.. ∗ (one or many), and 0.. ∗ (zero or many). The type of an attribute can be a type (e.g., ID, String, Boolean); an element type (e.g., the Actor of the isA attribute); or a vocabulary type (e.g., the ActorType of the type attribute).

The ActorType vocabulary rule is prefixed with the "enum" tag and defines the values of its parts, namely the literals 'User' and 'ExternalSystem'.

As shown in this simple example, a linguistic pattern defines two key aspects: a set of element attributes with respective name, type, and multiplicity; and a vocabulary, with a limited number of terms.

Furthermore, as suggested in Fig. 3, a linguistic pattern can be represented in multiple manners depending on the needs and interests of its users. In this context, a *linguistic style is a concrete representation of a linguistic pattern*, which means that a linguistic style is a specific template to which attributes of the linguistic pattern can be substituted. Thus, *a linguistic pattern can be represented by multiple linguistic styles*, such as CNL-A, CNL-B, and RSL.

A linguistic style is an ordered set of two types of text fragments: literal text fragments (e.g., 'actor', '[', ']', ', described as'); and other template text fragments that are represented by the pattern "<element_name.attribute_name>", i.e., the element name followed by its attribute name, delimited between '<' and '>' (e.g., <actor.type>). For brevity, it is possible to omit the reference to the "element_name.". The multiplicity constraints are represented by the same characters referred to above (i.e., '?', '+' and '∗').

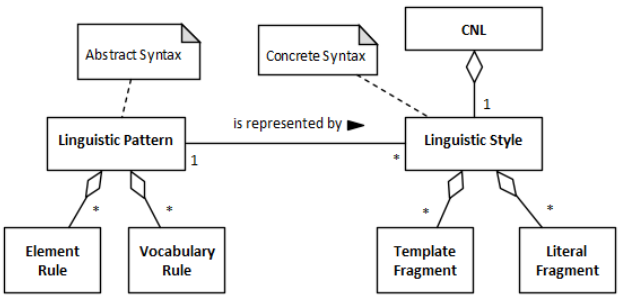

FIGURE 3. Relation between linguistic pattern and linguistic style (UML notation).

Also, the syntactical rules that define RSL constructs are compliant with the following general linguistic style, as a set of formal rules defined by an Extended BNF grammar. This grammar is supported by the Xtext framework (https://www.eclipse.org/Xtext/), which provides a ready-to-use integrated authoring tool, built on top of the Eclipse IDE.

```
'Element' id=ID (name=STRING )? ':' type=ElementType ('['
  ('[' 'isA' super=[Element])? ('description'
  description=STRING)? [etc.] ']')?
```

We can even use different formats to define linguistic styles. For example, using a verbose representation (i.e., using complete names of the attributes), we may have:

```
Actor <actor.id> [<actor.name>]? is a <actor.type> [,
  extends <actor.isA>]? [, associated with the
  Stakeholder <actor.stakeholder>]? [, described as
  <actor.description>]?.
```

Or using a more compact representation (i.e., using unqualified names of the attributes) we may have the equivalent representation. For the sake of simplicity, this would be the adopted format to represent the linguistic styles CNL-A and CNL-B throughout this paper.

```
Actor <id> [<name>]? is a <type> [, extends <isA>]? [,
  associated with the Stakeholder <stakeholder>]? [,
  described as <description>]?.
```

While for the definition of the RSL, we use the following type of representation:

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

IV. USE CASES' LINGUISTIC PATTERNS

As originally proposed by Jacobson, a use case is "a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal" [12]. Then, in the scope of the Use-Case 2.0 approach, Jacobson *et al.* define use case as "all the ways of using a system to achieve a particular goal for a particular user" [28]. Also, in the UML (Unified Modeling Language) specification, a use case is defined as "a set of behaviors performed by a subject, which yields an observable result that is of value for actors or other stakeholders of the subject" [29], in which "subject" is the system under discussion.

Some authors have adopted parts of the use case concept, notably Cockburn [15], Wirfs-Brock and Schwartz [16], and Constantine and Lockwood [17], [30] in what concerns the writing of use case scenarios. Also, use cases were adopted as a part of the UML and its diagrams are among the most widely used parts of the language [29]. However, despite its popularity, its adoption and use in practice are still hard and inconsistent [15], [28], [31]. Due to this situation, the patterns discussed in this paper consider the concept of "use case" as defined by the UML standard, i.e.., use cases to structure and specify the requirements of software systems, which means that actors interact directly with the system under consideration, or, as in Cockburn's terminology [15], they are "user-goals" or "elementary business processes" defined as "tasks performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state".

As illustrated in the mind map of Fig. 4, the linguistic patterns discussed in this paper involve the following: actor, use case, use case relationship, and use case scenario. The patterns actor, use case, and use case relationship are based on the UML language [29], while the pattern use case scenario is inspired by the previous work of Cockburn [15], Wirfs-Brock Schwartz [16], and Constantine and Lockwood [17]. The data entity and catch event patterns are used or referred in the scope of the proposed use case pattern, but are discussed elsewhere: in particular, the pattern data entity is extensively discussed by Silva and Savic [22], and the pattern catch event is based on the equivalent concept supported by languages like BPMN or UML activity diagram.

This paper uses a running example to support the explanation and discussion. This example refers to the requirements of a fictitious information system called "BillingSystem", which is an invoice management system inspired and adapted from the example initially introduced by Silva [21]. Fig. 5 depicts some use cases and relationships of this example with a simplified UML use case diagram: the BillingSystem has to support the management of invoices customers, users, products, etc. For the sake of simplicity, the examples showed in this section are only represented in CNL-A and CNL-B notations. However, Appendix A shows a more complete description of these specifications in CNL-A, CNL-B, and RSL, as well as an equivalent UML model.

A. ACTOR

1) LINGUISTIC PATTERN

In the scope of a requirements specification process, some elements suggest subjects that start or participate in interactions with the system under consideration. These elements might be referred to in multiple requirements (such as user stories or use cases) scattered throughout the text and, unfortunately, inconsistently used with different names. Usually, these elements denote different user roles, but can also denote

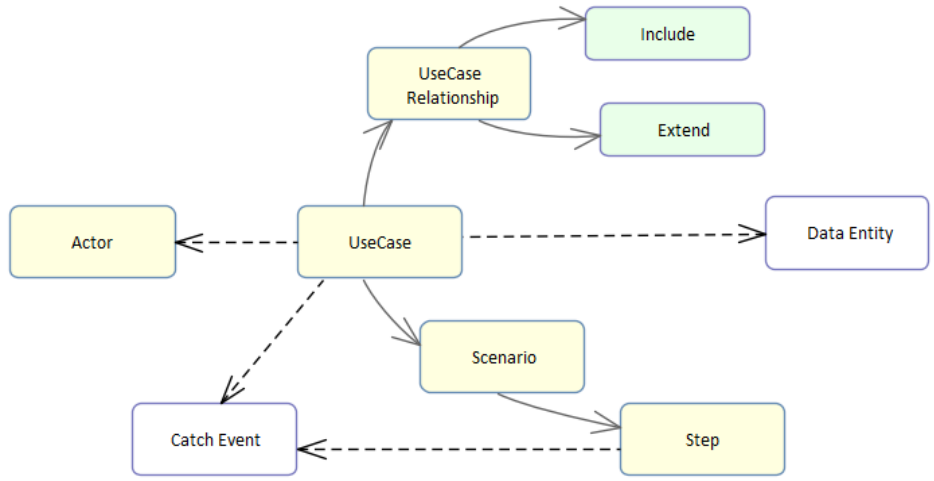

FIGURE 4. A top-level view of the Use Case's linguistic patterns (Mind Map notation).

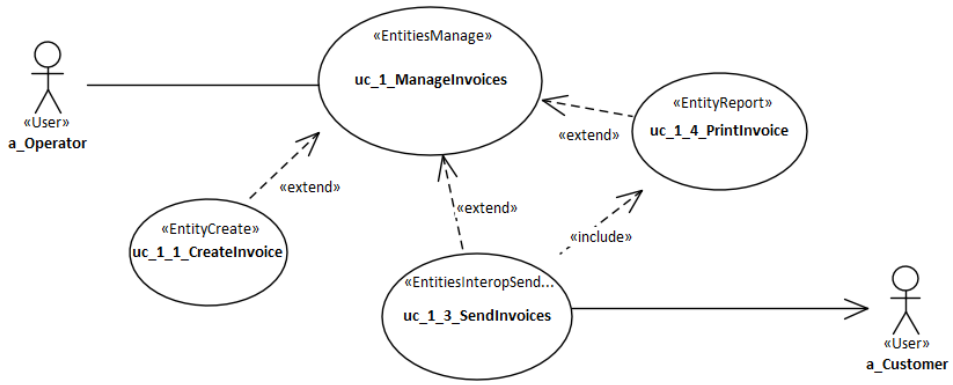

FIGURE 5. Use case diagram of the BillingSystem example (UML partial model).

external systems, with which the system interacts. A good practice is to explicitly define these elements with the Actor construct and refer to it by its unique identifier whenever relevant.

An Actor denotes an entity that has some goal on the use case. Actors are usually end-users or external systems that interact directly with the system and have a set of goals, i.e., tasks that need to get done using the system [12]. An actor is also a special stakeholder (that would have direct interaction with the system of discussion), but not every stakeholder is an actor (e.g., a ''manager'' actor is a stakeholder, while a ''project sponsor'' is a stakeholder but usually is not an actor because does not interact with the system.)

An Actor construct shall be defined by a unique id and a type. Optionally, it can also include a suggestive name, a generalization (isA) relationship to other (more abstract) actor, a reference to a stakeholder, and a general description. An actor shall be classified as a ''User'' or an ''ExternalSystem''. Besides, concerning the relationship between actors and use cases, the primary actor is the entity that has the use case's goal, while a supporting actor provides or receives some information or service to or from the system.

The following rules define the linguistic pattern Actor with its key fragments (lp1).

```
Actor ::
  <id:ID>  <name:String>? <type:ActorType>
  <isA:Actor>? <stakeholder:Stakeholder>?
  <description:String>?

enum ActorType ::
  User | ExternalSystem
```

The application of this pattern is shown in this paper with the languages CNL-A, CNL-B, and RSL. Also, this pattern is found in other languages such as the UML [29], SilabReq [32], [33], or XIS∗ [34]–[36]. However, these languages only support an id and name for the actors, while this pattern recommends the explicit classification of actors and the optional definition of the involved stakeholder.

2) LINGUISTIC STYLES

The following statements define different representations for this pattern:

Style according to the informal CNL-A (s1-cnl-a):

```
<id> is a <type> Actor [and a <isA>]? [, refers to
  <stakeholder>]?.
```

Style according to the informal CNL-B (s1-cnl-b):

```
Actor <id> [(<name>)]? is a <type> [, and a <isA>]? [,
  associated with the Stakeholder <stakeholder>]? [, who
  <description>]?.
```

Style according to RSL (s1-rsl):

```
'Actor' id=ID (name=STRING)?':' type=ActorType '['
  ('isA' super=[Actor])?
  ('stakeholder' stakeholder =[Stakeholder])?
  ('description' description=STRING)?
']'
```

3) GUIDELINES

In addition to a writing style, some practical guidelines shall be taken into consideration, namely concerning the writing of the following aspects:

- Id:
  1) *Identify the actor by a unique id* so that it can be easily referenced by use cases or in generalization relationships between actors. Some authors prefer to only use the name as the actor id for the sake of simplicity.
- Name:
  2) Name an actor as the user role or as the external system that interacts with the system. *Use the common role names that already exist; do not invent new ones*.
  3) *Do not use job titles* (e.g., "CEO", "CTO", "Project Sponsor") because, despite they may have specific needs and goals (i.e., they are stakeholders), they usually do not interact directly with the system.
- IsA (generalization relationship):
  4) *Only if need, define generalization relationships between actors*; this may be relevant if an actor is a generalization of others or vice-versa, i.e., a specialization of another more general actor.
- Stakeholder:
  5) *If relevant, reference the stakeholder associated with the actor*; this may help to explicitly establish relationships between actors and stakeholders. For instance, the actor "Customer" can be associated with the stakeholder "Organization Customer".
- Description:
  6) If relevant, briefly describe the main goals of the actor.

4) EXAMPLES

Considering the BillingSystem example (see Appendix A for a more complete description) and following a text analysis, we identify and annotate the actors with dashed underlined text and so, we can identify concepts like User, Administrator, Manager, Operator, Customer, and ERP. These shall be common names for user roles (e.g., Administrator, Manager, Operator, Customer) and external systems (e.g., ERP), but not names of concrete users (e.g., Mary or John), or job titles (e.g., CEO, CIO, project manager).

```
A user is someone that has a user account and is
assigned to a user role, namely as [...].
The manager shall [... ]. The operator shall [...].
The operator shall create invoices (with
respective details defined as invoice lines) [...].
While in the scope of the creation or update of an
invoice, the operator can create a customer record. The
approved invoices shall be sent to the respective
customer [...].
At the beginning of each year, the System shall archive
and export all paid invoices of the last year to the
external system ERP-System [...].
```

From this analysis and considering the linguistic styles defined above, actors can be defined as follows:

Actors represented according to the style s1-cnl-a:

```
a_Manager is a User Actor.
a_Operator is a User Actor.
a_Customer is a User Actor, refers to stk_Customer.
a_ERP is a ExternalSystem Actor.
```

Actors represented according to the style s1-cnl-b:

```
Actor a_Manager (Manager) is a User, who Approves
  invoices, etc.
Actor a_Operator (Operator) is a User, who Manages
  invoices and customers.
Actor a_Customer (Customer) is a User, associated with
  the Stakeholder stk_Customer, who Receives approved
  invoices to pay.
Actor a_ERP is a ExternalSystem.
```

B. USE CASES

A use case is defined as "a set of behaviors performed by a subject, which yields an observable result that is of value for actors or other stakeholders of the subject" [29]. As mentioned above in Section 1, this is a general definition [15], [28], [31]. So, it is difficult to consistently write use cases in practice. This means that additional guidance and information shall be associated with a use case, namely some classification schema, an involved data entity, etc. It shall be also important how to define relationships between use cases. Finally, it shall be important to define use case scenarios consistently.

To avoid the general definition of use cases, and consequent difficulty to write them, the use case construct shall involve the following aspects:

(1) A use case shall be classified by a set of use case types; for instance, types commonly found in information systems that support data management tasks, e.g., EntityCreate, EntityRead, EntityUpdate, EntityDelete actions, and more as suggested in Table 1.

(2) A use case may apply to a data entity or domain object as extensively discussed in [22].

(3) A use case shall define at least the primary actor (who has the goal or who triggers some action that led to achieving some goal), and optionally other supporting actors that might participate; these actors can be end-users or external systems (see Section in 4.1).

(4) By default, a use case is triggered by the primary actor; however, there are some situations in which use cases are triggered by events such as timers, receive messages, or conditional events.

(5) A use case can include preconditions and postconditions (or success guarantees) that define conditions or assumptions that shall be true, respectively, before and after a successful use case execution.

TABLE 1. Example of use case types.

UseCase Type	S/M Items (*)	Description
EntityCreate	S	Create a dataentity item
EntityRead	S	Read or consult a dataentity item
EntityUpdate	S	Update or edit a dataentity item
EntityDelete	S	Delete a dataentity item
EntityReport	S	Produce a report based on a dataentity item
EntityDashboard	S	Dashboard and data analysis based on a dataentity item
EntitiesManage	M	Manage (including CRUD operations) dataentity items
EntitiesBrowse	M	Browse and consult dataentity items
EntitiesReport	M	Produce a report based on multiple items of a dataentity
EntitiesDashboard	M	Dashboard and data analysis based on multiple items of a dataentity
EntitiesInterop Import	M	Import dataentity items from an external data source
EntitiesInterop Export	M	Export dataentity items to an external data source
EntitiesInterop Sync	M	Data Synchronization between dataentity items and an external data source
EntitiesInterop SendMessage	M	Send a message that includes dataentity items

* Applicable to a (S) single item or (M) multiple dataentity items

1) LINGUISTIC PATTERN

The following rules define the linguistic pattern UseCase (lp2).

```
UseCase::
   <id:ID> <name:String>? <type:UseCaseType>
   <stakeholder:Stakeholder>?
   <primaryActor:Actor>
   <supportingActors:Actor>*
   <triggeredBy:CatchEvent>?
   <dataEntity:DataEntityGeneric>?
   <precondition:STRING>?
   <postcondition:STRING>?
   ... (Use Cases Relationships, see Section 4.3)
   ... (Use Cases Scenarios, see Section 4.4)
   <description:String>?

enum UseCaseType::
  'EntityCreate' | 'EntityRead' | 'EntityUpdate' |
  'EntityDelete'[...] //see Table 1
```

The application of this pattern is shown extensively in this paper with the languages CNL-A, CNL-B, and RSL. Also, the application of this pattern is partially found in other languages such as UML [29] and SilabReq [32]. However, these languages do not provide some aspects and extensions proposed, such as the notion of use case classification; the relationship between a use case and a data entity used in its context; the relationship between a use case and the stakeholder that may have defined it; or the explicit identification of a catch event that may trigger the use case (e.g., a timer event), in those situations that the use case is not triggered by the primary actor.

2) LINGUISTIC STYLES

The statements below suggest different concrete representations for the UseCase linguistic pattern.

TABLE 2. Guidelines for writing use case's names: Common terms to use and to avoid.

Verb + Noun	Examples
Verb:	
Recommended	(Strong verbs) Create, Update, Merge, Calculate, Migrate, Send, Receive, Archive, Register, Activate, …
To avoid	(Weak or generic verbs) Make, Report, Use, Organize, Record, Find, Process, Maintain, List, …
Noun:	
Recommended	(Specific terms) Invoice, Payment, User account, Customer, …
To avoid	(Generic terms) Data, Paper, Report, System, Form, …

Style according to the informal CNL-A, with essential and few optional properties of a use case (s2-cnl-a):

```
<id> is a <type> UseCase [with <dataEntity>]? [, actor
  <primaryActor>] [ and participated by
  <supportingActors>*]? [, triggered by <triggeredBy>]?.
```

Style according to the informal CNL-B (s2-cnl-b):

```
UseCase <id> [(<name>)]? is a <type>
   [with <dataEntity>]?,
   [actor <primaryActor>]
   [and participated with actor(s) <supportingActors>*]?
   [, triggered by <triggeredBy>]?
   [, precondition <precondition>]?
   [, postcondition <postcondition>]?
   [, described as <description>]?.
```

Style according to RSL (s2-rsl):

```
'UseCase' id=ID (name=STRING)?':' type=UseCaseType '['
   ('primaryActor' primaryActor=[Actor])
   ('supportingActors' supportingActors=[Actor]*)?
   ('triggeredBy' triggeredBy=[CatchEvent])?
   ('dataEntity' dataEntity=[DataEntityGeneric]*)?

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

   ('description' description=STRING)?
']'
```

3) GUIDELINES

To better apply a writing style, some practical guidelines shall be considered, such as the following:

- Id:

  (G7) *Identify the use case by a unique id* to be easily referenced by other use cases (e.g., in the scope of include or extend relationships). Some authors prefer adopting a less techie format to express these ids (e.g., "manage invoices", "send invoices") while others prefer a format using only hierarchies of numbers (e.g., uc_1, uc_1_3) or a format combined by numbers and use case names (e.g., uc_1_manage_invoices, uc_1_3_SendInvoices). Regardless of your preference, *do apply your format consistently*.
- Name:

  (G8) Name a use case with a "verb-noun" structure that states the actor's goal. E.g., "Approve Invoices", "Send Invoices", "Register Payment".

  (G9) *Use concrete "strong" verbs instead of generalized, weaker ones* because weak verbs may indicate uncertainty. As summarized in Table 2, strong verbs

are: create, update, merge, calculate, migrate, send; while weaker verbs are: make, report, use, organize, record, find, process, maintain, list.

(G10) *Use specific nouns instead of generic terms* because specific terms are stronger. Examples of strong terms are invoice, payment, user account, customer; while weaker terms are: data, paper, report. In general, strong nouns correspond to the data entities' names.

(G11) *Define the verb from the primary actor's perspective*. For example, define "Receive Invoices not yet Paid" (stated from the actor manager's perspective) instead of "Send Invoices not yet Paid" (from the system's perspective); or "Receive Closed Invoices" (from the actor ERP's perspective) instead of "Send Closed Invoices" (from the system's perspective).

- Type:

  (G12) *Classify the use case according to some classification schema*; for example, based on types commonly found in information systems, like EntityCreate, EntityRead, EntityUpdate, EntityDelete, EntitiesImport, and more as suggested in Table 1. This can be useful to give a quick understanding of the use case and to support search and filter features on the management of use cases.
- Stakeholder:

  (G13)*Refer to the primary stakeholder that may have defined the use case*. Despite being optional, it can be relevant to identify who is the main stakeholder of the use case in case you need to understand and further discuss it.
- Primary Actor, Supporting Actors, and Trigger Event:

  (G14) The use case has a primary actor and may have one or more supporting actors. *In general and by default, consider the primary actor as the subject that triggers the use case* (e.g., "Create Invoice", "Register Payment") and so, in that situation *do not express any trigger event* because it would be redundant.

  (G15) Define and associate trigger events (e.g., a timer event "Beginning of the Year" or a conditional event "Invoices not paid after 30 days") to use cases that are not started by the primary actor but by the system based on some catch event or condition (e.g., "Receive Invoices not yet Paid", "Receive Closed Invoices").

  (G16) *Define and use consistently few types for classify trigger events; use only catch event types*, for example as defined by the BPMN language (e.g., with timer, conditional, signal, and error event types).
- Data entity:

  (G17) If relevant, *refer to the data entity manipulated in the scope of the use case*. In business information systems, use cases manipulate data entities, which are also commonly called conceptual or domain objects. For instance, "Invoice", "Customer" or "Product" are data entities of the running example (see [22] for further details).

4) EXAMPLES

Considering the BillingSystem example, candidate actors are annotated as dashed underlined text, use cases as underlined text, and data entities as **bold** text, as follows:

```
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. Also, for each invoice, the
user-operator needs to keep track if it is paid or not.
At the beginning of each year, the System shall archive
and export allpaid invoices of the last year to the
external system ERP-System [...].
```

From this analysis we identify the following use cases: "create a new invoice", "update invoice", "approve invoice", "issue and send invoice", "keep track if it is paid", and "archive and export all paid invoices". These use cases are mostly referring to a common data entity: invoice with its details. From this analysis and considering the linguistic styles defined above, we may define the following use cases specification:

Use Cases represented according to the style s2-cnl-a:

```
uc_1_ManageInvoices is EntitiesManage UseCase with
  e_Invoice, actor a_Operator.
uc_1_1_CreateInvoice is EntityCreate UseCase with
  e_Invoice, actor a_Operator.
uc_1_2_UpdateInvoice is EntityUpdate UseCase with
  e_Invoice, actor a_Operator.
uc_1_3_SendInvoices is EntitiesInteropSendMessage UseCase
  with e_Invoice, actor a_Operator and participated by
  a_Customer.
uc_1_4_PrintInvoice is EntityReport UseCase with
  e_Invoice, actor a_Operator.
uc_1_5_RegisterPayment is EntityUpdate UseCase with
  e_Invoice, actor a_Operator.
uc_4_ReceiveClosedInvoices is EntitiesInteropSendMessage
  UseCase with e_Invoice, actor a_ERP.
```

Use Cases represented according to the style s2-cnl-b:

```
UseCase uc_1_ManageInvoices (Manage Invoices) is a
  EntitiesManage with e_Invoice, actor a_Operator.
UseCase uc_1_1_CreateInvoice (Create Invoice) is a
  EntityCreate with e_Invoice, actor a_Operator.
UseCase uc_1_2_UpdateInvoice (Update Invoice) is a
  EntityUpdate with e_Invoice, actor a_Operator.
UseCase uc_1_3_SendInvoices (Send Invoices) is a
  EntitiesInteropSendMessage with e_Invoice, actor
  a_Operator and participated with actor aU_Customer.
UseCase uc_1_4_PrintInvoice (Print Invoice) is a
  EntityReport with e_Invoice, actor a_Operator,
  precondition ``e_Invoice.state in {Approved, Issued,
  Paid}''.
UseCase uc_1_5_RegisterPayment (Register Payment) is a
  EntityUpdate with e_Invoice, actor a_Operator,
  precondition ``e_Invoice.state = Approved'',
  postcondition ``e_Invoice.state = Paid''.
UseCase uc_4_ReceiveClosedInvoices (Receive Closed
  Invoices) is a EntitiesInteropSendMessage with
  e_Invoice, actor a_ERP, triggered by TimerEvent
  ``Beginning of the Year''.
```

C. USE CASES RELATIONSHIPS

According to the original definition of use cases [12] and as defined in UML [29], use cases may establish relationships to provide reusability and flexibility. To represent such relationships, the use case linguistic pattern shall consider the following additional aspects:

(1) A use case shall establish ''include'' relations to other use cases; this relation means that the behavior of the included use case is added to the source use case at a given point of its execution.

(2) A use case shall define ''extensions points'' in its context; this means that it can be extended by other use cases in these specific points and based on some extend relations.

(3) A use case shall extend the behavior of other use cases (the target use cases) in its specific extension point, by an ''extend'' relation.

1) LINGUISTIC PATTERN

The following rules extend the linguistic pattern UseCase (lp2) by supporting these relationships with the version (lp3).

```
UseCase::
    <id:ID> <name:String>? <type:UseCaseType>
    ...
    ('extensionPoints' <extensionPoints:ExtensionPoints>)?
    ('includes' <includes:UseCase>)*
    ('extends' <extends:UCExtends>)*
    ...

UCExtends:
    <targetUsecase:UseCase> 'onExtensionPoint'
  <targetExtensionPoint:ExtensionPoint>;
```

2) LINGUISTIC STYLES

The following statements define concrete representations for the extended UseCase linguistic pattern, but just for the CNL-B and RSL because, due to its inherent characteristics of a compact and simple writing style, CNL-A does not include such details of specification.

Style according to the informal CNL-B (s3-cnl-b):

```
UseCase <id> [(<name>)]? is a <type>
...
[, with extension points (<extensionPoints>)]?
[, includes <includes>]?
[, extends <targetUsecasen> on extension point
  <targetExtensionPointn>]?
...
```

Style according to RSL (s3-rsl):

```
'UseCase' id=ID (name=STRING)?':' type=UseCaseType '['
  ...
  (extensionPoints= UCExtensionPoints)?
  (includes= UCIncludes)?
  (extends+= UCExtends)*
 ...
```

3) GUIDELINES

To better apply a writing style, some practical guidelines shall be considered, such as the following:

- ExtensionPoints:
  (G18) *Define suggestive names for the extensions points;* these names represent optional features or functions that would extend the behavior of the source use case, which can *adapt the names of generic operations* (e.g., xp_Create, xp_Update, xp_Print) or *the names of the related use cases* (e.g., xp_PrintInvoice, xp_CreateCustomer); You may also prefix their names with a suggestive text, e.g. ''xp_'' or similar.
- Include relationship:
  (G19) Identify the use case (with its identifier) to establish an include relationship to another use case.
  (G20) *Do not concern with the relative order of the include relationship* because this shall only be expressed if you later define use case scenarios with respective steps (see Section 4.4).
- Extend relationship:
  (G21) Identify the use case (with a unique identifier) to establish an extend relationship with another use case (the target use case) and identify one of its extension points.
  (G22) *Do not concern with the relative order of the extend relationship* because this shall only be expressed if you define use case scenarios with respective steps.

4) EXAMPLES

Considering the BillingSystem example (see Appendix A) and the use cases defined previously in Section 4.2.4, we may define the relationships between them, namely ''extend'' and ''include'' relationships as depicted in the UML use case diagram of Fig. 5. For example, we may specify a general use case that supports the management of invoices (the uc_1_ManageInvoices) and defines several extension points (e.g., xp_Create, xp_Update, xp_ConfirmPayment). Then, the other more specific use cases (e.g., uc_1_1_CreateInvoice, uc_1_3_SendInvoices, and uc_1_4_PrintInvoice) establish an extend relationship with that general use case. On the other hand, considering that in the context of the use case ''send invoices'' (i.e., uc_1_3_SendInvoices) it needs to print the selected invoice(s) into a PDF file, this interaction can be specified by an include relationship between uc_1_3_SendInvoices and uc_1_4_PrintInvoice.

With these considerations in mind, we may redefine the above use cases, according to the linguistic style s3-cnl-b, as follows:

```
UseCase uc_1_ManageInvoices is a EntitiesManage with
  e_Invoice, actor a_Operator, with extensions points
  (xp_Create, xp_SendInvoices, xp_PrintInvoice).

UseCase uc_1_1_CreateInvoice is a EntityCreate with
  e_Invoice, actor a_Operator, extends
  uc_1_ManageInvoices on extension point xp_Create.

UseCase uc_1_3_SendInvoices is a
  EntitiesInteropSendMessage with e_Invoice, actor
  a_Operator and participated with actor a_Customer,
  extends uc_1_ManageInvoices on extension point
  xp_SendInvoices, includes uc_1_4_PrintInvoice.

UseCase uc_1_4_PrintInvoice is a EntityReport with
  e_Invoice, actor a_Operator, extends
  uc_1_ManageInvoices on extension point xp_PrintInvoice.
```

D. USE CASE SCENARIOS

Use cases can be further detailed with the specification of scenarios, namely one main or common scenario and several nested, alternative, and exception scenarios. There are several proposals for writing use case scenarios, however, we shall

consider the following aspects when reasoning about this linguistic pattern:

(1) A scenario can be described according to different formats, such as an informal narrative story or as a structured scenario with a flow of sequential steps, which is the format discussed in this paper.

(2) If relevant, the behavior of a use case can be detailed by a set of scenarios, which can be classified as Main, Nested, Alternative or Exception scenarios; a use case may have just one main scenario (also known as "basic path" or "happy flow"), and zero or more nested, alternative and exception scenarios; if relevant, a scenario may be described by a sequence of steps and additional sub-scenarios.

(3) Nested, alternative or exception scenarios shall be associated to one specific step, and so, can be considered sub-scenarios but with precise semantics, namely: (i) a nested scenario assigned to a specific step means that the behavior of such step is break-down into a sequence of (sub)steps performed as a nested sub-flow; (ii) an alternative scenario assigned to a specific step means that an optional or alternative behavior of that step can exist and can be defined with this scenario; (iii) an exception scenario means behavior that happens when something in that specific step can go wrong.

(4) Scenarios and steps shall be identified by a unique id in the scope of their use case, and this id shall suggest some nested and relative order, e.g., "1, 2, ..., 2.2, 2.3, 2.3a., ...".

(5) Steps are usually triggered by an actor or by the system under consideration, and, in more rare situations, by a catch event; consequently, they can be classified as Actor, System, or Event steps. Furthermore, a step can be classified by the involved type of action, e.g., Actor-PrepareData, Actor-CallSystem, System-Execute, System-ReturnResult, Event-Timer, Event-Conditional.

(6) There are special steps that refer to the behavior described by other use cases, which reflect include or extend relationships. In this situation, a step can express such behavior by a reference to the target or source use case and can use tags like "Include", "Extend" or "ExtendedBy (this corresponds to the inverse of the extend relationship).

1) LINGUISTIC PATTERN

The following rules define the linguistic pattern UseCaseScenario (lp4) that shall be considered in the scope of a use case.

```
UseCase::
   <id:ID> <name:String>? <type:UseCaseType>
   ...
   (<mainScenario:MainScenario>)?
   ...

MainScenario::
   (<idn:ID>)? <name:String>
   (<description:String>)?
   (<steps:Step>)*

Step::
   <id:ID>
   ((<type:StepType> (<actor:Actor>)? <name:String>) |
   (<ucTypen:StepUCType>  <usecase:UseCase>))
   (<description:String>)?
   (<nextStep:NextStep])?
   (<scenarios:Scenario>)*

Scenario::
   <id:ID>
   <type:ScenarioType>  <name:String>
   (<description:String>)?
   (<stepsn:Step>)*


enum ScenarioType:: 'Nested'|'Alternative'|'Exception'

enum StepType:: 'Actor' | 'System' | 'Event'

enum StepUCType:: 'Include' | 'Extend' | 'ExtendedBy'
```

2) LINGUISTIC STYLES

The following statements define two concrete representations for the UseCaseScenario linguistic pattern (lp4), namely for the languages CNL-B and RSL. Due to its inherent characteristics of a compact writing style, CNL-A does not include such details.

Style according the informal CNL-B (s4-cnl-b):

```
// UseCaseScenario
[<id>.]? Scenario <name> (Main): <description>?
<steps>*

// Step
<id>. [[<type [<actor>]?: <name>] | [<ucType>  <usecase>]]
[<description>]? .
[Rejoin at <nextStep:Step>]?.
<scenarios>*

// Scenario
<id>. Scenario <name> (<type>): [<description>]?
<steps>*
```

Style according RSL (s4-rsl):

```
'UseCase' id=ID (name=STRING)?':' type=UseCaseType '['
   ...
   (mainScenarios+=MainScenario)?
   ...

'mainScenario' id=ID '(' 'Main' ')' (name=STRING) '['
   ('description' description=STRING)?
   (steps+=Step)* ']';

'step' name=ID
   ('('(type=StepType)')'
  (name=STRING))|((typeUC=StepUCType)
  (usecase=[UseCase])))
   ('actor' actor=[Actor])?
   ('description' description=STRING)?
   ('nextStep' next=Step)?
   ('[' (scenarios += Scenario)* ']' )?

'scenario' id=ID '(type=ScenarioType) (name=STRING) '['
   ('description' description=STRING)?
   (steps+=Step)* ']';
```

3) GUIDELINES

Follows practical guidelines for writing use case scenarios:

- Scenarios and Use Cases:

  (G23) For each use case, *determine the common (or "happy") path to the actor's goal*; Ignore other possible paths through the use case at first; write these scenarios later, as alternative or exception subscenarios.

  (G24) *Write a scenario as a sequence of steps, ordered by time*.

  (G25) *If a subscenario appears complex* or if *it is important* and you want to emphasize it, you may decide to *document it as a separate and complementary*

use case and express such connection through (i) an include relationship, if it is a nested scenario; or (ii) an extend relationship (or the inverse ''extendedBy'' relationship) if it is an alternative or exception scenario. E.g., the step ''3. Actor: Select a customer'' of the uc_1_1_CreateInvoice has an alternative scenario, 3a. Scenario CustomerNotExist, which expresses an include to the use case uc_1_6_CreateCustomer.

(G26) *Do not use both extend and extendedBy relationship types; pick one type and use it consistently*. For instance, in this paper we used ''include'' and ''extendedBy'' in the examples shown (see Appendix A).

- Scenarios:

(G27) *Do not give an id to the main (or common) scenario*; however, if you still want to give it an id, use some special value like ''0'', ''Step.0'' or so.

(G28) *Define an id to uniquely identify a nested, alternative, or exception scenario*; this id shall be combined with the step id *to which the scenario is assigned* (e.g., ''3.'', ''Step3.'') with a letter (e.g., ''a'', ''b'') giving an id such as ''3.a'', ''Step2.b''.

(G29) *Define a name to a scenario*; for example, ''MainScenario'' for the common scenario and a short readable name like ''SelectAndUnselectItems'', ''SearchItems'', ''CustomerNotSelectedException'', for the other scenarios.

(G30) *Optionally, define a brief description of the scenario*; e.g., ''Select and unselect invoices'', ''Search invoices'', ''Customer not yet selected''.

(G31) *Define the type of the scenario*; namely: ''Main'' for the main or common scenario, and ''Nested'', ''Alternative'' or ''Exception'' for, respectively, nested, alternative, or exception subscenarios.

(G32) *Be consistent in how to begin and end the scenarios*: i.e., your scenarios shall begin and end by a step triggered by the system or by the actor. For instance, in our examples, the main scenarios started always with a step triggered by the system, e.g., the first step of the uc_1_1_CreateInvoice's main scenario is ''1. System: Shows the interaction space CreateInvoice ...'', but could be started by the actor with ''1. Actor: Select the option CreateInvoice from the interaction space ManageInvoices''. Both options are acceptable, so, pick one and be consistent throughout your specification.

(G33) *Do not use ''Rejoint at'' for nested scenarios*: A ''Rejoint at'' is a special property of a step that allows to terminate or interrupt the current control flow by specifying the id of the next step; Because a nested scenario represents a set of sub-steps, when its last sub-step ends the control flow resumes to the next top-level step, so you do not need to express that in a nested scenario.

(G34) *Do use ''Rejoint at'' for alternative or exception scenarios, but only if need*, i.e., only when the last sub-step (of the alternative or exception scenario) ends the control flows do not resume to the next top-level step.

- Steps:

(G35) *Define explicitly the type of the step* that shall reflect who triggers the action: in general, the actor or the system. Because you have explicit the step type, *do not add to the step sentence again the name ''actor'', ''system'' or similar* because it will become redundant. E.g., ''3. Actor: Select a customer'', ''4. System: Shows the customer's data'' in the main scenario of uc_1_1_CreateInvoice.

(G36) For those *use cases that are triggered by events* (e.g., see uc_3_ReceiveAlertOfInvoicesNotYetPaid or uc_4_ReceiveClosedInvoices), *their first steps are usually of type ''event''*. Use consistently the same types as discussed for G16, for instance, as defined by the BPMN language (with timer, conditional, signal, and error event types). *Define a consistent format to write event-based steps*, e.g., for timer events, ''Event Timer: Beginning of the year'' or ''Event Timer: Starts on 1st of January at 21:00, repeat every year'' based on a template like ''starts on <date> [at <time>] [, repeat every <time-period>]''.

(G37) *Write a step with a simple explanatory statement, with no branching, exceptions, or groups of steps*; if need use instead nested, alternative or exception scenarios. See e.g., the nested scenario of the uc_1_1_CreateInvoice ''5a: Scenario CreateInvoiceLine (Nested)'', which is assigned to step 5, defined as a simple statement like ''5. Actor: Creates one or more invoice lines''.

(G38) *Write steps in consistent and easy-to-read statements; Steps shall not describe details related to user interfaces, preliminary design, software architecture, or quality requirements*. Avoid the use of techie terms and *define your own set of verbs and objects*, like those shown in Table 3, and use them consistently.

4) EXAMPLES

Considering the BillingSystem example (see Appendix A) and the use cases defined previously in the previous sections, we may consider that the main scenario of the use case ''uc_1_1_CreateInvoice'' is defined informally by the following brief description:

```
MainScenario of uc_1_1_CreateInvoice (Create Invoice)

The System shows a list of Invoices and available
  actions, namely of CreateInvoice, UpdateInvoice,
  ConfirmPayment, SendInvoices, and PrintInvoice. In
  addition, there are actions to Close the interaction
  space, Select/Unselect Invoices, Search Invoices, and
  Filter Invoices...
The actor Operator fills the form, with the following
  sub-steps:
The actor selects the customer field
The actor creates one or more InvoiceLines ...
The actor can change the invoice's date creation field
The actor triggers the action Create
The System validates the submitted data and creates the
  Invoice (with respective InvoiceLines)
The System returns a feedback success message "Invoice
  Created and Waiting for Approval".
```

TABLE 3. Guidelines for writing steps: Common verbs and terms to use and to avoid.

Class of terms	Examples
Verbs triggered by the System:	
Recommended	Show, Display, Create, Save, Ask, Check, Validate, Send, Export, Import, …
To avoid	Show Web Page, Create X object, Save to Database, HTTP POST, GET API, …
Verbs triggered by an Actor:	
Recommended	Fill, Select, Set, Enter, Confirm, Receive, …
To avoid	Click, Double-click, Drag-Drop, Scroll, Tap, Swipe, Pinch, ...
Verbs triggered by an Event:	
Recommended	Start on, Trigger, Trigger when, …
To avoid	Happen, Occur, ...
UI related terms:	
Recommended	User interface, UI, Interaction Space, Dialog, UI Element, Action, Option, …
To avoid	Page, Form, Dialog-box, Label, Text box, Check box, List box, Menu, Menu Option, Link, Button, …

However, this informal description can be improved significantly, for instance, according to the linguistic style s4-cnl-b and the discussed guidelines:

UseCase uc_1_1_CreateInvoice

```
[...]
0. Scenario MainScenario (Main):
1. System: Shows the interaction space CreateInvoice with
  empty fields and with the following actions: Confirm,
  Cancel, AddInvoiceLine, AddCustomer; by default, the
  invoice's creation date is filled with the current
  date, and the invoice's id is filled with an auto-
  number.
2. Actor: Selects a customer
3. System: Shows the customer's data, namely its full
  name and fiscal identification
4. Actor: Creates one or more invoice lines
5. Actor: Selects the action Confirm.
6. System: Validates the submitted data.
7. System: Creates the Invoice information, which shall
  be in the ''Pending'' state.
8. System: Displays a success message of ''Invoice
  Created and Waiting for Approval''.

2a. Scenario CustomerNotExist (Alternative): Customer
  does not exist and has to be created
   2a.1. Actor: Selects the action AddCustomer
   2a.2. [ExtendedBy uc_1_6_CreateCustomer].

4a: Scenario CreateInvoiceLine (Nested): Create invoice
  lines [Repeat ''1.. *'']
   4a.1. Actor: Selects the action AddInvoiceLine
   4a.2. Actor: Selects a product and enters the number
   of items
   4a.3. System: Updates the InvoiceLine and Invoice
   values.
...
6a: Scenario CustomerNotSelectedException (Exception):
  Customer not yet selected
  6a.1 System: Check that a customer was not yet
  selected.
  6a.2 System: Displays an error message.
  Rejoin at 2.
6b: Scenario InvoiceLineNotAddedException (Exception):
  Invoice lines not yet added
  6b.1 System: Check that there is not at least one
  invoice line added.
  6b.2 System: Displays an error message.
  Rejoin at 4.
```

Further examples are included in Appendix A with the scenarios of the use cases depicted in Fig. 5.

V. RELATED WORK

Writing requirements specifications has involved creating descriptions of the application domain, prescription of what the system shall do, and other organizational, legal, or technological constraints [1], [2], [5], [6]. In general, these requirements have been specified with natural languages due to their higher expressiveness and ease of use. However, the use of natural language presents also disadvantages like ambiguity, inconsistency, and incompleteness, extensively discussed in the literature [1], [9]–[11]. Due to these problems, specifications are usually written in a natural language but complemented by some sort of models and other representations that use controlled natural languages (CNLs) or semi-formal modeling languages.

A CNL is a constructed language based on a natural language, being more restrictive concerning lexicon, syntax, and semantics while preserving most of its natural properties [25]. CNLs have been designed to support technical writing or knowledge engineering, to improve the translation, or to provide natural and intuitive representations for formal notations. CNLs can be classified into two general categories: human-oriented and machine-oriented CNLs. Human-oriented CNLs intend to improve the readability and comprehensibility of technical documentation and to simplify and standardize human-human communication for special purposes. On the other hand, machine-oriented CNLs intend to improve the translatability of technical documents and the acquisition, representation, and processing of knowledge. Since the structure of CNLs is usually simpler than the structure of natural language, in general, CNLs are easier for a computer to process and more natural for humans to understand. An ideal CNL for knowledge representation should also be effortless to write and expressive enough to describe the problem at hand [37].

These languages (e.g., CNLs and modeling languages) include a set of elements – like glossary terms, data entities, actors, use cases, or other types of requirements – that explicitly or implicitly define its abstract syntax and semantics, and addresses different concerns at multiple abstraction levels. Also, these languages offer different notations or concrete syntaxes such as textual, graphical, tabular, or even form-based representations [38].

Our first attempts on the research of requirements specifications started with the design of languages by Videira and Silva [13], [14], Ferreira and Silva [39], Savic *et al.* [33], Ribeiro and Silva [35], [36], and recently Silva [26] with the RSL language. RSL is a rigorous CNL for the specification of requirements and tests, that includes constructs commonly used by the community, such as goals, use cases, user stories, constraints, or quality requirements [21], [27]. The design of

RSL strongly influenced (and was influenced by) the proposal of this set of patterns.

As referred in Section 1, this is the third on a series of papers that discuss best practices for the writing of requirements specifications. The first paper of this series [21] introduces the notions of linguistic pattern and linguistic style and proposes a set of patterns for business-level concepts, such as glossary term, stakeholder, business task, event, and flow. It also discusses the representation of such patterns with the RSL language. The second paper of this series [22] discusses a set of patterns for data entities and related elements (e.g., data attributes, constraints, and enumerations) and introduces the CNL-A and CNL-B informal languages. That second paper also presents an extensive discussion of the related work by comparing different representations of data-related elements, specifically, by comparing NLs, formal languages, CNLs, and modeling languages based on the PENS framework [25]. Apart from the introduction of the notions of linguistic pattern and linguistic style, that are shared among these papers, this third paper has the same goals as the others, but it is focused on how to better write use cases and scenarios. Still, and differently from the others, this paper also provides an extensive number of practical guidelines.

Use case is one type of requirement that attracted a lot of interest from the community, specifically in the scope of modeling languages like UML [29] or SysML [40]. Use cases were originally proposed by Jacobson to describe the functional requirements of a software system in a text paragraph style format [12]. Jacobson's use cases were deliberately simple and informal so that they could be understood by both technical and business stakeholders. Their simplicity makes them a popular technique for the aim of documentation and validation [9], [28], [41].

But, as mentioned above, the use of natural languages present difficulties in what concerns consistency and understandability [1], [2], [9]–[11], [22]. That means that without recommendations on writing styles and specific guidance it would be difficult to produce high quality and consistent specifications. In this respect, Table 4 shows a concise comparison of relevant work in this area of requirements specifications focused on use cases. For each work (i.e., each line of the table), ''Y'' indicates the presence of a specific property or contribution. Each work contributes at a document and or a sentence level and, at each of these levels, can contribute with linguistic patterns (P), linguistic styles (S), and guidelines (G), as discussed throughout this paper. Finally, involving document-level approaches, a work can propose and discuss aspects related to file templates (T).

For a more extensive analysis, Tiwari and Gupta [41] review the existing literature for the evolution of the use cases, their applications, quality assessments, open issues, and future directions. They performed a keyword-based search to identify the most relevant studies related to use case specifications research, which resulted in a data set of 119 papers published between 1992 and 2014.

A. DOCUMENT-LEVEL APPROACHES

Some authors have discussed *how to organize* requirements at a *document-level*, usually in the form of ''requirements specification templates''. These requirements templates are structured in chapters and complementary appendixes and are usually supported by templates in Word, Excel, or similar files, which are then customized or used directly.

For instance, Robertson and Robertson [42] propose the Volere requirements specification template that first appeared in 1995 and has evolved since then. The Volere techniques are described in their book ''Mastering the Requirements Process'' [2] and have expanded to include several aspects such as the context or scope model, business use case, and product use case scenarios to model responses and provide a basis for both story writing and implementation strategy.

Cockburn [15] discusses some general aspects for a requirements specification template (or in his terminology, a ''requirements file''), based on simple customization of the Volere template.

Silva *et al.* [8] survey some requirements templates – specifically IEEE Std 830-1998 [10], RUP Software Requirements Specification Template [3], and Withall template [6] – and discuss a set of guidelines that would allow structuring a requirements specification template in chapters and complementary appendixes minimizing their dependencies.

Currently, web repositories like TemplateLAB(https://templatelab.com) or Project Management Docs(https://www.projectmanagementdocs.com) provide a variety of use case templates and examples in Word or Excel files, some of them inspired on the original contributes of Robertson and Robertson [42], Constantine and Lockwood [17], Cockburn [15] and others. However, these templates usually only provide a general structure and format for such documents and do not include (or just include in a shallow way) practical guidance on how to better write the sentences included in those documents.

B. SENTENCE-LEVEL APPROACHES

Conversely, some works have discussed how to write requirements specifications at *the sentence-level* more systematically and consistently, which is the perspective of this paper. Of course, these two groups of approaches – document-level and sentence-level – shall be combined to produce better requirements specifications.

1) APPROACHES ON USE CASES AND SCENARIOS

In what concerns the writing of use cases and scenarios there have been several contributes. Some guidelines and recommendations have been proposed namely concerning the structure and style of sentences to reduce ambiguities and inconsistencies, such as CP [19], CREWS [43], CR [44], and SSUCD's [45]. For instance, CREWS provides six style guidelines and eight content guidelines for writing and refining these scenarios, and for integrating different scenarios into a use case [46].

TABLE 4. Summary of the Related Work.

Paper			Document-level		Sentence level								
					Use cases			Scenarios			Other Requirements		
Reference	Authors	Title	T	G	P	S	G	P	S	G	P	S	G
This Paper					**Y**	**Y**	**Y**	**Y**	**Y**	**Y**			
[8]	Silva et al. (2014)	Towards a System Requirements Specification Template that Minimizes Combinatorial Effects	Y	Y									
[42]	Robertson & Robertson (2012)	Volere Requirements Specification Template	Y	Y									
[21]	Silva (2017)	Linguistic Patterns and Linguistic Styles for Requirements Specification (I): An Application Case with the Rigorous RSL/Business-level Language									Y	Y	
[22]	Silva and Savić (2021)	Linguistic Patterns and Linguistic Styles for Requirements Specification (II): Focus on Data Entities									Y	Y	
[18]	Lilly (1999)	Pragmatic cures for the writing of use cases			Y		Y						
[17]	Constantine and Lockwood (2001)	Essential use cases			Y	Y		Y	Y				
[15]	Cockburn (2001)	Writing Effective Use Cases	Y		Y	Y	Y	Y	Y	Y			
[16]	Wirfs-Brock (2001)	The art of writing use cases				Y	Y		Y	Y			
[50]	Mavin et al. (2009)	Easy approach to requirements syntax (EARS)									Y	Y	Y
[11]	INCOSE (2019)	Guide for Writing Requirements, Version 3										Y	Y

(T)emplates; (G)uidelines; (P)atterns; (S)tyles

Tiwari and Gupta [41] survey twenty use case templates (at both document and sentence levels) that have been proposed and applied for various specification problems ranging from informal descriptions with paragraph-style text to more formal keyword-oriented templates. Examples of such templates are proposals by Lilly [18], Constantine and Lockwood [17], Wirfs-Brock and Schwartz [16], Leite *et al.* [47], Cockburn [15], Dutoit and Paech [48], or Kulak and Guiney [49]. Tiwari and Gupta [41] conclude that use cases have been evolved from initial plain, semi-formal textual descriptions to a more formal template structure facilitating automated information extraction in various software development life cycle activities such as requirement documentation, requirement analysis, requirement validation, domain modeling, test case generation, planning and estimation, and maintenance.

However, these use case templates show some limitations that are addressed and mitigated to some extent with our proposal. First, they tend to mix both the ''concepts'' with their ''representations'' or, in our terminology, the linguistic patterns (i.e., abstract syntax with the definition of the key elements and respective properties) with the linguistic styles (i.e., concrete syntax with textual representations). In general, most of the proposals are more focused on the discussion of the concepts and not on their representation. For example (i) the original use case template, proposed by Robertson and Robertson [42], included elements like use case name, preconditions, postconditions, basic and alternate flow of events; or (ii) the use case template proposed by Cockburn [15] included fields such as scope, level of abstraction, trigger, main and alternate flows, variations, and extension points.

Second, these concepts and respective representations are usually characterized in a general and informal way (e.g., with just natural language descriptions complemented with some simple examples), which raises difficulties with their application and use in practice. On the contrary, in our paper, both linguistic patterns and linguistic styles are defined with formal rules that help to mitigate these difficulties. In addition, by providing that separation between the concepts and their representations, anyone can extend or define their representations of use cases and scenarios but keeping and applying the same set of concepts.

Third, there are some concepts introduced in the linguistic patterns discussed in this paper that were not found in the analysis of the related work, namely the concepts of (i) actor and use case classification, (ii) use case associate with a data entity, (iii) use case triggered by a catch event, (iv) scenario and step classification, (v) scenario that can be started by a catch event or actor or the system itself.

Fourth, some guidelines discussed in this paper were naturally borrowed from the related work, such as: Identify the actor and the use case by a unique id (G1 and G7); Name a use case with a ''verb-noun'' structure that states the actor's goal (G8); For each use case, determine the common (or ''happy'') path to the actor's goal; Ignore other possible paths through the use case at first; write these scenarios later, as alternative or exception sub-scenarios (G23); Write a scenario as a sequence of steps, ordered by time (G24). However, others were not found from the literature analysis, namely, to mention a few: Classify the use case according to some classification schema (G12); In general and by default, consider the primary actor as the subject that triggers the use case (G14); Define and associate a trigger event to use cases that are not started by the primary actor but by the system based on some event or condition (G15); Define and use consistently few types for classify trigger events; use only catch event types (G16); If relevant, refer to the data entity manipulated in the scope of the use case (G17).

2) OTHER APPROACHES

Concerning the writing of other types of requirements, some approaches have influenced the research discussed in this paper. For instance, the ''INCOSE Guide for Writing Requirements'' [11] is a popular and respected reference in the systems engineering community, which includes a broad set of characteristics and rules for helping write clear and concise requirements and needs. The INCOSE guide provides forty-one rules organized in the following categories: accuracy, concision, non-ambiguity, singularity, completeness, uniqueness, abstraction, tolerance, uniformity, and modularity. Despite this guide mainly consider functional requirements, some rules can apply as well to the writing of use cases, e.g., Define terms (R4), Refer to diagrams and tables (R25), Use active voice and identify the actor (R2), Classify requirements by type (R31), or Conform to structure and patterns (R44).

Mavin *et al.* [50] propose five structural rules for the writing of functional requirements based on the EARS approach (the Easy Approach to Requirements Syntax), namely rules for the following classes of requirements: ubiquitous (i.e., with the general structure ''The <System Name> shall <System Response>''), state-driven, event-driven, optional feature, and unwanted behavior.

On the other hand, from the agile software development community, user stories have been widely used as elements to represent the requirements from the user's point of view. A user story is a semi-structured specification of requirements using a sentence template following a popular form like [51]: ''As <Who>, I want/need/can/would like <What>, so that <Why>'', where it defines Who wants it, What is expected from the system, and optionally, Why it is important [51], [52]. Despite their popularity, user stories are usually defined in a general and high-level way, with each main action captured in a separate story. Therefore, stakeholders may lose the ''bigger picture'' of the system they are developing while the number of stories increases [53]. Moreover, some proposals have discussed similarities between user stories and use cases, and how to provide transformations between them [54], [55].

Raharjana *et al.* [56] conduct a systematic literature review on the analysis and impact of NLP (natural language processing) tools on user stories and concluded that the purpose of NLP studies in user stories is broad, ranging from discovering defects, generating software artifacts, identifying the key abstraction of user stories, and tracing links between model and user stories. Many of these findings are similarly applicable to other types of requirements such as use cases or functional requirements. For instance, manually reviewing requirements against such a large number of rules and guidelines, as discussed above, is a tedious, time-consuming, and error-prone task. To reduce these limitations, this process has been supported by software tools, like RAT (https://www.reusecompany.com), QVscribe (https://qracorp.com/qvscribe/), IBM RQA (https://www.ibm.com/products/requirements-quality-assistant), or Requirements Scout (https://www.qualicen.de/en/), which use NLP techniques to automatically analyze and check the quality of requirements against some of these guidelines, like EARS [50], [57] and INCOSE [11], [58], [59]. Arrabito *et al.* [60] report an experience using three NLP analysis tools, concerning their performance in detecting ambiguity and under-specification in requirements documents and compare them concerning other qualities like learnability, usability, and efficiency.

VI. EVALUATION BASED ON A PILOT USER SESSION

To evaluate the patterns, styles, and guidelines proposed in this paper, and to receive feedback from people not directly involved in this research, we conducted a pilot user session based on a questionnaire that was sent to several subjects (mainly IT students, researchers, professionals, and colleagues work on the RE area) and answered during April and May of 2021. That questionnaire was answered by a group of 24 subjects (see Fig. 6 for a graphical demographic analysis), 8 women and 16 men, with at least a BSc degree, namely 7 with a BSc, 9 with an MSc, and 8 with a Ph.D. degree. Most of the participants had professional experience, namely 4 participants with less than 3 years, 3 participants between 3 and 10 years, and 10 participants with more than 10-years of experience. Participants had previous experience in the following IT roles: software developer (10), requirements engineer (9), business analyst (8), quality assurance engineer, and software tester (2).

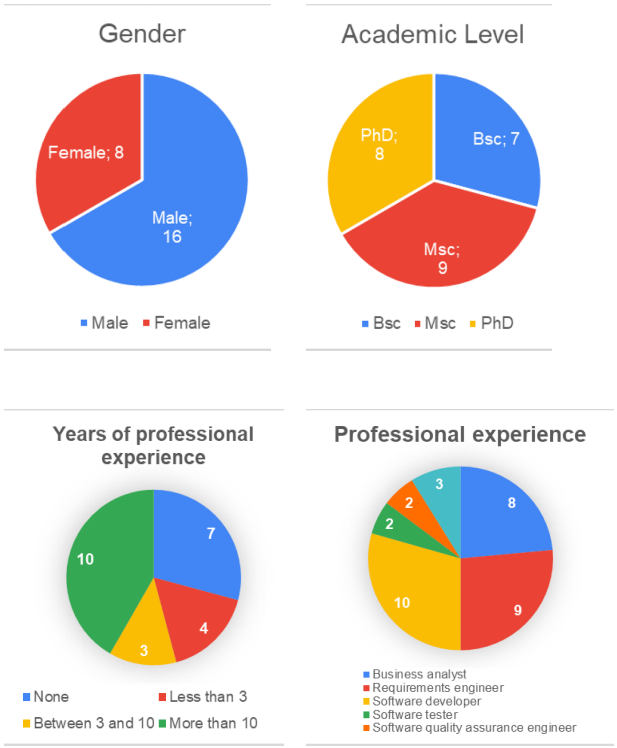

FIGURE 6. Demographic analysis of the group of participants in the pilot user session.

A. USER SESSION SETUP

The user session was conducted under the following conditions: tests took place at the participant's environment (e.g., office or home environment); realization of the task without previous use and learning of the proposed patterns; main information sent by email; users were free to think and share ideas if they wanted.

The user session and respective preparation processes involved the following tasks:

Task-1: (Preparation) We prepared a PDF document (available at https://bit.ly/2Sxs1UE) that described a draft version of the proposed linguistic patterns, styles, and guidelines; this document also included a shorter version of the "BillingSystem" example as discussed in the paper.

Task-2: (Preparation) We prepared a questionnaire using Google Forms (available at https://bit.ly/3xPZt8Y) consisting of 3 sections with a total of 17 questions:

Section 1: Respondent Characterization. The first five questions (Q1 to Q5) were focused on the general characterization of the participant with the following aspects: nationality, gender, academic level, years of professional experience, and type of professional experience.

Section 2: Linguistic Patterns, Styles, and Guidelines. Nine questions (Q6 to Q14) were directly related to the assessment of the proposed patterns, styles, and guidelines; we first asked participants to rate in a 5-Likert scale (i.e., from 0 to 5, 0—Do not know, 1-Very Low, 2 -Low, 3-Medium, 4-High, and 5-Very High) how does she rate the proposed linguistic patterns, linguistic style, and guidelines, in what concerns different specific qualities (e.g., relevance, simplicity, expressiveness, readability, completeness). Then, we include open-ended questions so that respondents could provide more information and feedback based on their knowledge and understanding.

Section 3: Further Feedback. Three additional questions (Q15 to Q16) were more time-consuming: the participant was challenged to specify some use cases informally referred in the case study (and not included in the sent PDF), namely specify the use cases "uc_2_ApproveInvoices", "uc_2_1_ApproveInvoice", "uc_3_AlertInvoicesNotYet Paid". Finally, participants were invited to shortly explain their decisions regarding the previous question and provide their general feedback.

Task-3: (Preparation) We prepared a list of participants and invited them to collaborate in the user session. An email with clear instructions on how to complete the survey was sent to these candidate participants and they were asked to fill in the evaluation questionnaire.

Task-4: (Execution) Each participant read the PDF document with the additional instructions and performed autonomously the proposed tasks; in the end, they answered the evaluation questionnaire.

Task-5: (Analysis) We collected the responses submitted by the participants and analyzed their results.

B. QUESTIONNAIRE ANALYSIS

As referred above, the questionnaire has several questions grouped into three sections, in which section 2 is the most extensive and involves three dimensions of analysis.

The first dimension concerns the evaluation of the discussed *linguistic patterns* (i.e., Actor, Use Case, Use Case Relationships, and Use Case Scenario) based on the following question: Q6. How do you rate the discussed Linguistic Patterns? Responses to this dimension revealed good to very good grades, as summarized in Table 5 and illustrated in the box plot of Fig. 7. Use Case Relationships pattern was rated slightly below 4 (i.e., 3.96), while the others had the average ratings above 4, namely 4.04 for Actor, 4.26 for Use Case, and 4.13 for Scenario patterns in the 5-Likert scale (in which 1 is Very Low and 5 Very High). There was one participant that answered 0 (meaning that she did not know how to rate these aspects), and we did not consider her answer for these numbers.

Some of the feedback provided by the participants on the open-ended question about linguistic patterns was: "my suggestion is to use figures for each example (use case, actor, etc.) because the reader having a quick look at the figures can better understand the involved concepts", "they capture well the use case specs", "linguistic patterns are useful to deal with the challenges of using natural language to describe requirements", "you are asking for two different things in your questionnaire question: (a) relevant and (b) complete".

The second dimension evaluates the proposed *guidelines* associated with the respective patterns (i.e., Actor, Use Case,

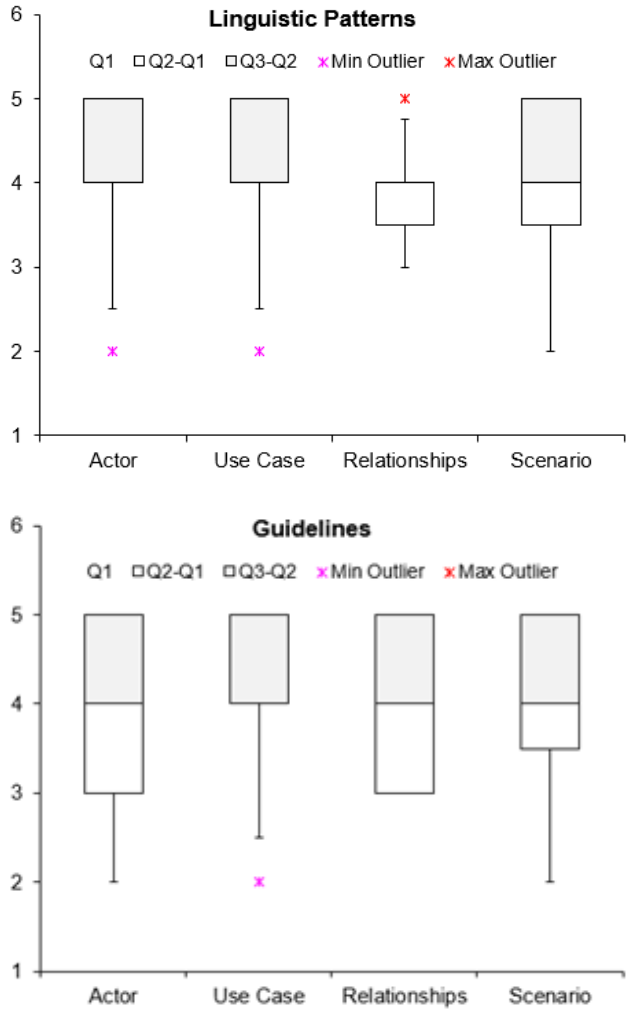

FIGURE 7. Results from the survey on the linguistic patterns (above) and guidelines (below), box plots.

TABLE 5. How do participants rate the linguistic patterns and Guidelines?.

Linguistic Patterns	Patterns	Guidelines
Actor	4,04	3,96
Use Case	4,26	4,17
Use Case Relationships	3,96	4,00
Use case Scenario	4,13	4,17

Use Case Relationships, and Use Case Scenario), which is based on the following question: Q8. How do you rate the Guidelines of the discussed linguistic patterns? Responses to this question also revealed good to very good grades, as summarized in Table 5 and illustrated in the box plot of Fig. 7. Guidelines to the writing of the Actor pattern was rated slightly below 4 (i.e., 3.96), while the others had the average ratings above or equal 4, i.e., 4.00 for Use Case Relationships, and 4.17 for Use Case and Scenario, in the 5-Likert scale (in which 1 is Very Low and 5 Very High).

TABLE 6. How do participants rate the linguistic Styles (values on a 1–5 scale)?.

Styles	Precision	Expressiveness	Naturalness	Simplicity	Completeness
CNL-A	3,46	3,17	3,42	3,96	3,30
CNL-B	3,67	3,88	3,75	3,58	3,83
RSL	4,25	3,67	2,96	3,08	4,08

Some of the feedback provided by the participants on the open-ended question about the guidelines was: ''they are quite good and promote the quality of use case specs'', ''use case relationships are less likely to create ambiguities and mistakes since there are not many things that can go wrong, so the guidelines are not as relevant as the others'', ''the guidelines are relevant and complete''.

The third dimension evaluates the *writing styles* CNL-A, CNL-B, and RSL considering the following qualities: simplicity, expressiveness, readability, and completeness. Table 6 summarizes the average scores on this dimension, based on which we may verify the following findings: RSL is the most precise and complete writing style, with a score of, respectively, 4.25 and 4.08, but is the less natural and simple language, with respectively a score of 2.96 and 3.08. On the other hand, CNL-A is less precise (with 4.46) and complete (3.30). Finally, between these two, RSL-B is the most expressive (3.88) and natural (3,75) writing style. Still regarding this third dimension, to the question Q13 (i.e., Which notation do you consider more appropriate for writing use cases and scenarios?), 12 (50%) participants responded that CNL-B is the most appropriate notation, followed by 10 participants (42%) that voted for RSL, and only 2 participants (8%) that voted for CNL-A. Follows some feedback provided by the participants in this respect: ''CNL-B is easier to read and understand'', ''the CNL-B seems to be a good middle ground between too much simplicity (such as provided by CNL-A) and too much complexity (such as provided by RSL)'', ''CNL-B is a more natural language, so easier to learn and adopt, although on a second run I consider the other notations to be more precise and therefore more helpful to someone that is going to implement the specified requirements'', ''CNL-B provides a balance of detail and succinct'', ''RSL seems to be the most suitable'', ''CNL-B might be a little easier to understand since it's closer to the natural languages, however RSL gives more complete information, while maintaining the simplicity to read all the information'', ''I prefer RSL because CNL-A and CNL-B are less complete but globally more near natural language; as you insert more details, RSL is more far from natural language but it is more precise and complete'', ''I prefer CNL-B because we need to discuss the use cases

with the domain experts; the more the language deviates from natural language, the harder it will be for them to validate the requirements''. These users' preferences are aligned with what we had perceived since RSL sentences are less natural and less easy to read than the equivalents in CNL-A or CNL-B. On the other hand, CNL-B provides a good balance between expressiveness, naturalness, and simplicity.

Regarding the open-ended questions of section 3, most of the participants did not answer. However, those that answered provide encouraging comments and feedback, such as: ''Good work, Congrats!'', ''Keep going the work, really good'', ''The paper gives some very nice explanations on the 'notion' of a use case and summarizes a lot of very helpful do's and dont's; as a lecturer (which I am right now) I would love to share them with my students (hence, I am longing for your paper to be published!); as a business analyst, working quite extensively with use case (10+ years ago, for 10+year), I do see a huge benefit in the do's and dont's''. The analysis of these comments led to the conclusion that it would be helpful for the participants to have an appropriate tool editor that would better support the writing of the chosen style, and, on the other hand, they refer that more research shall be developed around user stories and agile approaches.

To sum up, the results collected from this evaluation were generally encouraging with positive scores in all the analyzed dimensions. Even when it is stated that the number of participants in the session is small, some usability experts, like Nielsen and Landauer, observed that a group of 5 testers is enough to uncover over 80% of the usability problems [61]. Also, since our questionnaire focuses on the usability of the patterns, styles, and guidelines, we may claim that 24 participants are a fair number for an exploratory assessment, at least to identify major flaws in the usability of such proposals.

VII. CONCLUSION

A requirements specification describes the technical concerns of a system and is used throughout several stages of its life cycle. It is a technical document that helps to share a common vision of the system under consideration among the main stakeholders, as well as facilitate its development and management processes. The use of natural language in such specifications is widespread and effective because humans can easily read and write them. However, natural language also shows inherent characteristics that often present as the root cause of many quality problems, such as inconsistency, incompleteness, and ambiguousness.

Due to these problems, natural language specifications are many times supplemented with some sort of other models and representations that use controlled, formal or semi-formal modeling languages. These languages may include a set of constructs (e.g., data entity, actor, use case, scenarios, user stories) that explicitly or implicitly define its abstract syntax and semantics, and addresses different concerns and abstraction levels. These languages also provide different notations or concrete syntaxes, such as textual, graphical, tabular, form-based representations. Additionally, as discussed in the related work, there are some proposals to better write different types of requirements, such as functional requirements [2], [10], [11], [50], [57]–[59], user stories [51], [52], [56], or use cases and scenarios [15], [16]–[18], [47]–[49]. For instance, Tiwari and Gupta [41] survey twenty use case templates that have been proposed and applied for various specification problems.

However, these ''use case templates'' have some limitations that are addressed and mitigated to some extent with our proposal: First, they tend to mix both the ''concepts'' with their ''representations''. Second, these concepts and respective representations are usually defined in a general and informal way, which raises difficulties with their application and use in practice. Third, there are some concepts introduced in the linguistic patterns discussed in this paper that were not found in the related work, namely: the concepts of actor and use case classification, use case associate to a data entity, use case triggered by a catch event, scenario and step classification, a scenario that can be started by a catch event or actor or the system itself. Fourth, some guidelines discussed in this paper were naturally borrowed from the related work, however, others were not found from the related work, such as: Classify the use case according to some classification schema (G12); In general and by default, consider the primary actor as the subject that triggers the use case (G14); Define and associate a trigger event to use cases that are not started by the primary actor but by the system based on some event or condition (G15); Define and use consistently few types for classify trigger events; use only catch event types (G16); If relevant, refer to the data entity manipulated in the scope of the use case (G17).

We claim that we need better and more specific guidance to improve the way requirements engineers and product designers write and validate their specifications. We argue that improving the awareness and knowledge of linguistic patterns, like the ones discussed in this paper, complemented with writing styles and practical guidelines, may contribute to enhance this current situation. On the other hand, as showed throughout this paper with the languages CNL-A, CNL-B, and RSL, these linguistic patterns may be represented in practice by different writing styles, being it compact (as with CNL-A), verbose (as CNL-B), or even based on a rigorous requirements specification language (as with RSL). This discussion was supported by a simple yet pedagogical and straightforward running example that allows illustrating, in fact, both the proposed linguistic patterns and associated writing styles. Appendix A includes a more complete version of this example with multiple textual representations as well as a visual UML notation.

This work was validated with a pilot user session with 24 participants, IT professionals and students, not directly involved in the research. The feedback received from this experiment was generally promising with positive scores in all the analyzed dimensions.

For future work, we intend to research the following challenges. First, survey how RE practices are adopted in

real-world projects by IT consulting organizations, and evaluate how close these practices are from the contributes of this paper. Second, discuss and define further linguistic patterns, namely for user stories, goals, other types of requirements, as well as for test cases and acceptance criteria. Third, explore and evaluate different writing styles, including and combining textual with tabular and visual representations. Fourth, explore and implement document automation features, specifically by automatically generate requirements specifications according to different writing styles (e.g., compact or verbose styles) from an intermediate format like the one defined with RSL or similar. Fifth, research NLP techniques and libraries, and applied them to automatically check and improve the quality of requirements at sentence and document levels.

APPENDIX A: THE RUNNING EXAMPLE: BILLINGSYSTEM

This paper uses a running example to support the explanation and discussion of the proposed linguistic patterns and linguistic styles. This example refers to the requirements of a fictitious information system called "BillingSystem" which is an invoice management system. The following text partially describes a variety of its informal requirements. For the sake of consistency, we slightly modify the example initially introduced in [21].

A. INFORMAL REQUIREMENTS

The following sentences describe the informal requirements of the BillingSystem. For the sake of readability, these sentences highlight special text fragments as follows: candidate actors are dashed underlined text; data entities are **bold**, and use cases are marked as underlined text.

The BillingSystem is a system that allows its users to manage **customers**, **products**, and **invoices**.
A user is someone that has a **user account** and is assigned to a **user role**, namely as administrator, manager, operator, and customer.
The administrator may register and manage **users**[...].
The manager shall manage **products** and shall configure **VAT** (value-added tax) values [...].
The operator shall manage **customers** and **invoices** [...].
The operator shall create **invoices** (with respective details defined as **invoice lines**). An **invoice** shall have the following information: customer id, dates (e.g., of creation, approval, and paid), status (e.g., created, approved, sent, paid), total value with and without VAT. Also, an **invoice line** shall include product id, number of items, product value with and without VAT. While in the scope of the creation or update of an **invoice**, the operator can create a **customer** record.
Before sending an invoice to a customer, the **invoice** shall be formally approved by the manager. Only after such approval, the operator shall print and send that **invoice** electronically by e-mail. In addition, for each **invoice**, the operator needs to keep track if it is paid or not.
The System shall automatically alert the manager, for all the **invoices** sent to customers but not yet paid, after 30 days of their respective issue date.
At the beginning of each year, the System shall archive and export all paid **invoices** of the last year to an external system, the ERP-System.
[...]

The following text fragments describe the use cases and related elements represented with the CNL-A, CNL-B, and RSL languages. For the sake of briefness, these fragments present a complete specification for only the use cases most directly related to the management of invoices. Also, for the sake of understandability and further comparison, Fig. A.1, in Section A.2, shows the equivalent UML use case diagram, and Section A.3 provides a very compact writing style based on the user story pattern.

B. REPRESENTED GRAPHICALLY WITH A UML USE CASE DIAGRAM

C. WITH AN AGILE AND VERY COMPACT WRITING STYLE

```
DataEntities: e_Customer, e_Invoice, e_InvoiceSimple, ...

Actors: Manager, Operator, Customer, ERP, ...

User Stories:
As a Operator I want to Manage invoices.
As a Operator I want to Create an invoice.
As a Operator I want to Update an invoice.
As a Operator I want to Send invoices to the customer.
As a Operator I want to Print invoices.
As a Operator I want to Register the payment of paid
  invoices.
As a Operator I want to Create customer records.
As a Manager I want to Consult invoices waiting for
  approval.
As a Manager I want to Approve invoices.
As a Manager I want to Receive alert of invoices not paid
  When (conditional event) Invoices not paid after 30
  days.
As a ERP system I want to Receive closed invoices When
  (timer event) Beginning of the year.
```

D. WITH A COMPACT WRITING STYLE (CNL-A)

```
DataEntities:
e_Customer is a Master DataEntity [... ].
e_Invoice is a Document DataEntity [... ].
e_InvoiceSimple is a Document DataEntity [...].


Actors:
a_Manager is a User Actor.
a_Operator is a User Actor.
a_Customer is a User Actor.
a_ERP is a ExternalSystem Actor.


Use Cases:
uc_1_ManageInvoices is a EntitiesManage UseCase with
  e_Invoice, actor a_Operator.
uc_1_1_CreateInvoice is a EntityCreate UseCase with
  e_Invoice, actor a_Operator,
   extends uc_1_ManageInvoices.
uc_1_2_UpdateInvoice is a EntityUpdate UseCase with
  e_Invoice, actor a_Operator,
   extends uc_1_ManageInvoices.
uc_1_3_SendInvoices is a EntitiesInteropSendMessage
  UseCase with e_Invoice, actor a_Operator and
  participated by a_Customer, includes
  uc_1_4_PrintInvoice, extends uc_1_ManageInvoices.
uc_1_4_PrintInvoice is a EntityReport UseCase with
  e_Invoice, actor a_Operator,
   extends uc_1_ManageInvoices.
uc_1_5_RegisterPayment is a EntityUpdate UseCase with
  e_Invoice, actor a_Operator,
   extends uc_1_ManageInvoices.
uc_1_6_CreateCustomer is a EntityCreate UseCase with
  e_Customer, actor a_Operator,
   extends uc_1_1_CreateInvoice and uc_1_2_UpdateInvoice.
uc_2_ConsultInvoicesToApprove is a EntitiesBrowse UseCase
  with e_Invoice, actor a_Manager.
uc_2_1_ApproveInvoice is a EntityUpdate UseCase with
  e_Invoice, actor a_Manager,
   extends uc_2_ConsultInvoicesToApprove.
```

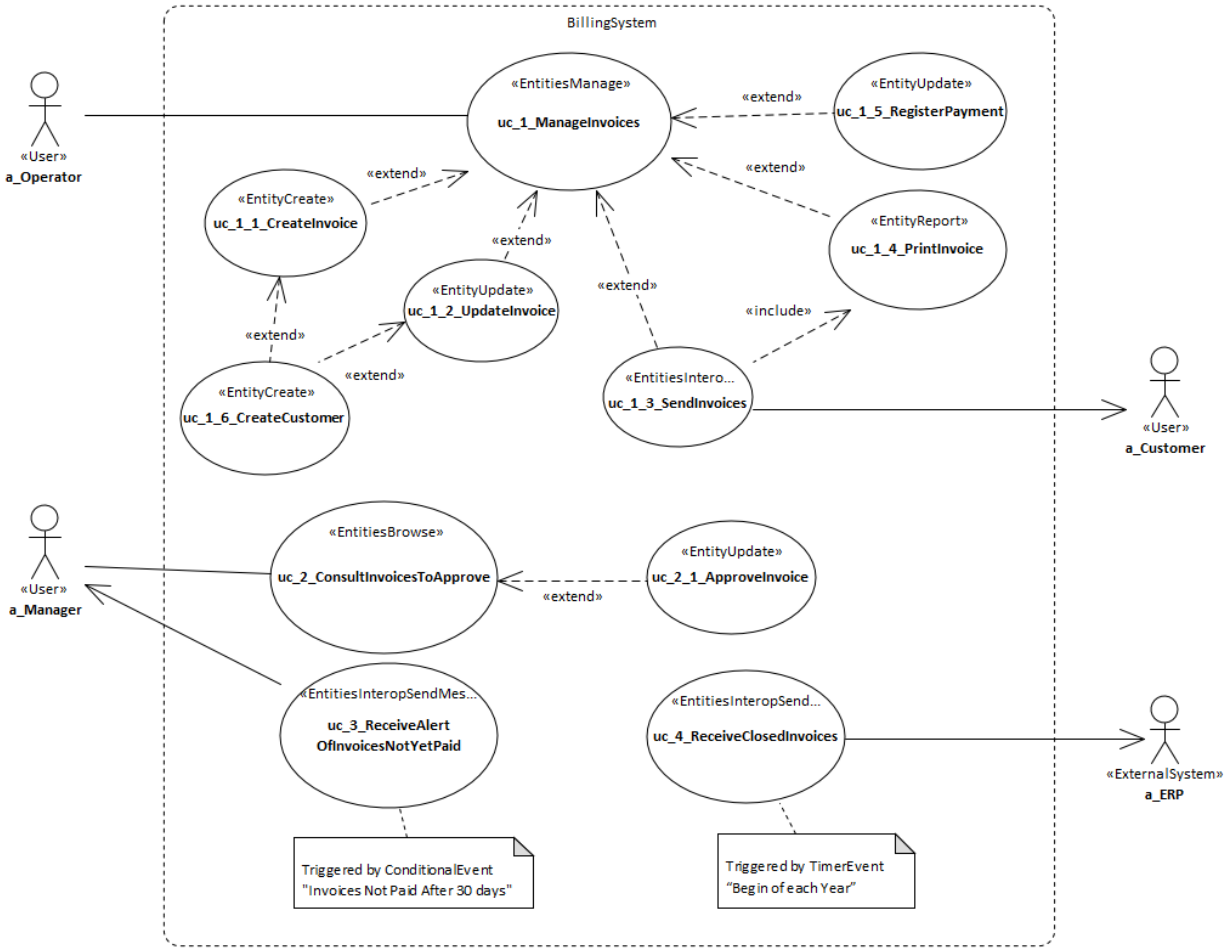

FIGURE A.1. Partial use case model of the BillingSystem (UML notation).

uc_3_ReceiveAlertOfInvoicesNotYetPaid is a
EntitiesInteropSendMessage UseCase with e_Invoice,
actor a_Manager, triggered by ConditionalEvent
''Invoices not paid after 30 days''.
uc_4_ReceiveClosedInvoices is a
EntitiesInteropSendMessage UseCase with
e_InvoiceSimple, actor a_ERP, triggered by TimerEvent
''Beginning of the Year''.

E. WITH A VERBOSE WRITING STYLE (CNL-B)

DataEntities:
DataEntity **e_Customer** is a Master [...].
DataEntity **e_Invoice** is a Document [...].
DataEntity **e_InvoiceSimple** is a Document [...].

Actors:
Actor **a_Manager** (Manager) is a User, who Approves
invoices, etc.
Actor **a_Operator** (Operator) is a User, who Manages
invoices and customers.
Actor **a_Customer** (Customer) is a User, associated with
the Stakeholder stk_Customer, who Receives approved
invoices to pay.
Actor **a_ERP** is a ExternalSystem.

Use Cases:
UseCase **uc_1_ManageInvoices** is a EntitiesManage with
e_Invoice, actor a_Operator, with extension points
(xp_Create, xp_Update, xp_ConfirmPayment,
xp_SendInvoices, xp_PrintInvoice).
UseCase **uc_1_1_CreateInvoice** is a EntityCreate
with e_Invoice, actor a_Operator,
postcondition ''Invoice.state = Pending'',
extends uc_1_ManageInvoices onExtensionPoint
xp_Create, with extension points (xp_CreateCustomer).
UseCase **uc_1_2_UpdateInvoice** is a EntityUpdate with
e_Invoice, actor a_Operator,
extends uc_1_ManageInvoices onExtensionPoint xp_Update,
with extension points (xp_CreateCustomer).
UseCase **uc_1_3_SendInvoices** is a
EntitiesInteropSendMessage with e_Invoice, actor
a_Operator and participated with actor a_Customer,
postcondition ''Invoice.state = Issued'',
includes uc_1_4_PrintInvoice,
extends uc_1_ManageInvoices onExtensionPoint
xp_SendInvoices.
UseCase **uc_1_4_PrintInvoice** is a EntityReport with
e_Invoice, actor a_Operator,
precondition ''Invoice.state in Approved, Issued,
Paid'',
extends uc_1_ManageInvoices onExtensionPoint
xp_PrintInvoice.
UseCase **uc_1_5_RegisterPayment** is a EntityUpdate with
e_Invoice, actor a_Operator,
precondition ''Invoice.state in Issued'',
postcondition ''Invoice.state = Paid'',
extends uc_1_ManageInvoices onExtensionPoint
xp_ConfirmPayment.
UseCase **uc_1_6_CreateCustomer** is a EntityCreate with
e_Customer, actor a_Operator,
extends uc_1_1_CreateInvoice onExtensionPoint

```
  xp_CreateCustomer,
  extends uc_1_2_UpdateInvoice onExtensionPoint
  xp_CreateCustomer.
UseCase uc_2_ConsultInvoicesToApprove is a EntitiesBrowse
  UseCase with e_Invoice, actor a_Manager,
  with extension points (xp_ApproveInvoice).
UseCase uc_2_1_ApproveInvoice is a EntityUpdate with
  e_Invoice, actor a_Manager,
  precondition ''Invoice.state = Pending'',
  postcondition ''Invoice.state = Approved OR
  Invoice.state = Rejected'',
  extends uc_2_ConsultInvoicesToApprove onExtensionPoint
  xp_ApproveInvoice.
UseCase uc_3_ReceiveAlertOfInvoicesNotYetPaidis a
  EntitiesInteropSendMessage with e_Invoice, actor
  a_Manager, triggered by ConditionalEvent ''Invoices not
  paid after 30 days''.
UseCase uc_4_ReceiveClosedInvoices is a
  EntitiesInteropSendMessage with e_InvoiceSimple, actor
  a_ERP, triggered by TimerEvent ''Beginning of the
   Year''.
```

Use Case' Scenarios:

UseCase uc_1_ManageInvoices

```
[...]
0. Scenario MainScenario (Main):
1. System: Shows a list of Invoices and available
  actions, namely of CreateInvoice, UpdateInvoice,
  ConfirmPayment, SendInvoices, and PrintInvoice. In
    addition, there are actions to Close the interaction
  space, Select/Unselect Invoices, Search Invoices, and
  Filter Invoices.
2. Actor: Browses the list of Invoices and consult
  Invoices
3. Actor: Selects the option Close.
4. System: Shows the interaction space Home.
2a. Scenario FilterItems (Alternative): [...]
2b. Scenario SearchItems (Alternative): [...]
2c. Scenario SelectAndUnselectItems (Alternative): Select
  and unselect invoices [...]
2d. Scenario CreateInvoice (Alternative):
  2d.1. Actor: Selects the action CreateInvoice.
  2d.2. [ExtendedBy uc_1_1_CreateInvoice] Creates an
  Invoice.
2e. Scenario UpdateInvoice (Alternative):
  2e.1. Actor: Selects the action UpdateInvoice.
  2e.2. [ExtendedBy uc_1_2_UpdateInvoice] Updates a
  selected Invoice.
2f. Scenario SendInvoices (Alternative):
  2f.1. Actor: Selects the action SendInvoices.
  2f.2. [ExtendedBy uc_1_3_SendInvoices] Sends selected
  invoices to a customer.
```

UseCase uc_1_1_CreateInvoice

```
[...]
0. Scenario MainScenario (Main):
1. System: Shows the interaction space CreateInvoice with
  empty fields and with the following actions: Confirm,
  Cancel, AddInvoiceLine, AddCustomer; by default, the
  invoice's creation date is filled with the current
  date, and the invoice's id is filled with an auto-
  number.
2. Actor: Selects a customer
3. System: Shows the customer's data, namely his full
  name and fiscal identification
4. Actor: Creates one or more invoice lines
5. Actor: Selects the action Confirm.
6. System: Validates the submitted data.
7. System: Creates the Invoice information, which shall
  be in the 'Pending' state.
8. System: Displays a success message of 'Invoice Created
  and Waiting for Approval'.

2a. Scenario CustomerNotExist (Alternative): Customer
  does not exist and has to be created
  2a.1. Actor: Selects the action AddCustomer
  2a.2. [ExtendedBy uc_1_6_CreateCustomer].
4a: Scenario CreateInvoiceLine (Nested): Create invoice
  lines [Repeat ''1.. *'']
4a.1. Actor: Selects the action AddInvoiceLine
4a.2. Actor: Selects a product and enters the quantity
  of items
4a.3. System: Updates the InvoiceLine and Invoice
  values.

6a: Scenario CustomerNotSelectedException (Exception):
  Customer not yet selected
  6a.1 System: Check that a customer was not yet
  selected.
  6a.2 System: Displays an error message.
  Rejoin at 2.

6b: Scenario InvoiceLineNotAddedException (Exception):
  Invoice lines not yet added
  6b.1 System: Check that there is not at least one
  invoice line added.
  6b.2 System: Displays an error message.
  Rejoin at 4.
```

UseCase uc_1_3_SendInvoices

```
[...]
Scenario MainScenario (Main):
1. System: Shows the interaction space SendInvoices,
  namely with the following settings [...], and with
  available actions: Send, Cancel.
2. Actor: Sets criteria to send invoices [...], namely
  shall select just one Customer and the respective
  Invoices to send.
3. Actor: Select the action Send.
4. System: Validates the send criteria.
5. System: Creates an email message with all selected
  invoices.
6. System: Send an email message to the Customer.
7. System: Displays a success message of 'Invoices Sent
  to the Customer'.

5a: Scenario PrintAndAttachInvoices (Nested): Print and
  Attach Invoices
  1. [Include uc_1_4_PrintInvoice] [Repeat for each
  selected invoice].
  2. System: Attach the printed PDF files to the email
  message.
```

UseCase uc_1_6_CreateCustomer

```
[...]
Scenario MainScenario (Main):
1. System: Shows the interaction space CreateCustomer,
  with empty fields, and with available actions (Confirm,
  Cancel); by default, the customer's creation date is
  filled with the current date.
2. Actor: Fills the customer fields [...].
3. Actor: Select the action Confirm.
4. System: Validates the submitted data.
5. System: Creates the Customer information.
6. System: Shows the previous interaction space.
```

UseCase uc_4_ReceiveClosedInvoices

```
[...]
Scenario MainScenario (Main):
1. Event Timer: Beginning of each year.
2. System: Produces the list of closed invoices from
  e_Invoice.
3. System: Sends the list of closed invoices to the
  a_ERP.
4. Actor: Receives the list of closed invoices.
5. System: Moves the closed invoices from e_Invoice to
  e_ClosedInvoice.
6. System: Register the success operation in Log.
```

F. WITH A RIGOROUS WRITING STYLE (RSL)

DataEntities:

```
DataEntity e_Customer: Master [...]
DataEntity e_Invoice ''Invoices'': Document [...]
DataEntity e_Invoice_Simple: Document [...]
```

Actors:

```
Actor a_Manager ''Manager'': User [description ''Approves
  Invoices, etc.'']
Actor a_Operator ''Operator'': User [description
  ''Manages invoices and customers'']
Actor a_Customer ''Customer'': User [description
  ''Receives approved invoices to pay'']
```

```
Actor a_ERP: ExternalSystem

Use Cases:
UseCase uc_1_ManageInvoices ''Manage Invoices'':
  EntitiesManage [
  primaryActor a_Operator
  dataEntity ec_Invoice
  extensionPoints xp_Create, xp_Update,
  xp_ConfirmPayment, xp_SendInvoices,
xp_Print]

UseCase uc_1_1_CreateInvoice ''Create Invoice'':
  EntityCreate [
  primaryActor a_Operator
  dataEntity e_Invoice
  postcondition ''e_Invoice.state = Pending''
  extensionPoints xp_CreateCustomer
  extends uc_1_ManageInvoices onExtensionPoint xp_Create]

UseCase uc_1_2_UpdateInvoice ''Update Invoice'':
  EntityUpdate [
  primaryActor a_Operator
  dataEntity e_Invoice
  extensionPoints xp_CreateCustomer
  extends uc_1_ManageInvoices onExtensionPoint xp_Update]

UseCase uc_1_3_SendInvoices ''Send Invoices'':
  EntitiesInteropSendMessage [
  primaryActor a_Operator
  supportingActors a_Customer
  dataEntity e_Invoice
  postcondition ''e_Invoice.state = Issued''
  includes uc_1_4_PrintInvoice
  extends uc_1_ManageInvoices onExtensionPoint
  xp_SendInvoices]

UseCase uc_1_4_PrintInvoice ''Print Invoice'':Entity
  Report[ primaryActor a_Operator
  dataEntity e_Invoice
  precondition ''e_Invoice.state in Approved, Issued,
  Paid'' extends uc_1_ManageInvoices onExtensionPoint
  xp_Print]

UseCase uc_1_5_RegisterPayment ''Register Payment'':
  EntityUpdate [
  primaryActor a_Operator
  dataEntity e_Invoice
  precondition ''e_Invoice.state = Issued''
  postcondition ''Invoice.state = Paid''
  extends uc_1_ManageInvoices onExtensionPoint
  xp_ConfirmPayment]

UseCase uc_1_6_CreateCustomer ''Create Customer (in the
  Invoice context)'': EntityCreate [
  primaryActor a_Operator
  dataEntity e_Customer
  extends uc_1_1_CreateInvoice onExtensionPoint
  xp_CreateCustomer
  extends uc_1_2_UpdateInvoice onExtensionPoint
  xp_CreateCustomer]

UseCase uc_2_ApproveInvoices ''Approve Invoices'':
  EntitiesBrowse [
  primaryActor a_Manager
  dataEntity e_Invoice
  extensionPoints xp_ApproveInvoice]

UseCase uc_2_1_ApproveInvoice ''Consult and Approve
  Invoice'': EntityUpdate [
  primaryActor a_Manager
  dataEntity e_Invoice
  precondition ''e_Invoice.state = Pending''
  postcondition ''e_Invoice.state in Approved,
  Rejected''
  extends uc_2_ApproveInvoices onExtensionPoint
  xp_ApproveInvoice]

UseCase uc_3_ReceiveAlertOfInvoicesNotYetPaid:
  EntitiesInteropSendMessage [
  primaryActor a_Manager
  actorParticipates a_Manager
  triggeredBy ConditionalEvent ''Invoices not paid after
  30 days''
  dataEntity e_Invoice_Simple]

UseCase uc_4_ReceiveClosedInvoices:
  EntitiesInteropSendMessage [
  primaryActor a_ERP
  triggeredBy TimerEvent ''Starts on 1st of January at
  21:00, repeat every year''
  dataEntity e_Invoice_Simple
  precondition ''e_Invoice.state = Paid'']

Use Case' Scenarios:

UseCase uc_1_ManageInvoices
[...]
mainScenario s0 (Main) ''Happy Flow'' [
step s1 (System) ''Shows a list of Invoices and available
  actions, namely of CreateInvoice, UpdateInvoice, ...''
step s2 (Actor) ''Browse the list of Invoices and consult
  Invoices'' [
  scenario s2a (Alternative) ''Filter invoices'' [
    step s1 (Actor) ''Selects the option 'aFilter'''
    step s2 (System) ''Shows the filter options''
    step s3 (Actor) ''Sets the criteria to filter
    invoices''
    step s4 (Actor) ''Selects the option Filter''
    step s5 (System) ''Shows a list of invoices that
    satisfy the defined filter criteria'' ]
scenario s2b (Alternative) ''Search invoices'' [
    step s1 (Actor) ''Select the option a Search''
    step s2 (System) ''Shows the search options''
    step s3 (Actor) ''Enters the search query''
    step s4 (Actor) ''Select the option Search''
    step s5 (System) ''Shows a list of invoices that
satisfy the defined search criteria'']
scenario s2c (Alternative) ''Select and unselect
invoices'' []
scenario s2d (Alternative) ''Create invoice'' [
   step s1 (Actor) ''Selects the option CreateInvoice''
   step s2 <extendedBy uc_1_1_CreateInvoice> ]
scenario s2e (Alternative) ''Update invoice'' [
   step s1 (Actor) ''Selects the option UpdateInvoice''
   step s2 <extendedBy uc_1_2_UpdateInvoice> ]
scenario s2f (Alternative) ''Send invoices'' [
   step s1 (Actor) ''Selects the option SendInvoices ''
   step s2 <extendedBy uc_1_3_SendInvoices> ] ]
step s3 (Actor) ''Selects the option Close''
step s4 (System) ''Shows the interaction space Home'' ]


UseCase uc_1_1_CreateInvoice
[...]
mainScenario s0 (Main) ''Happy Flow'' [
step s1 (System) ''Shows the interaction space
  CreateInvoice with empty fields and
   with the following actions: Confirm, Cancel,
  AddInvoiceLine, AddCustomer;
   by default, the invoice's creation date is filled with
  the current date, and
   the invoice's id is filled with an auto-number.''
step s2 (Actor) ''Selects a customer'' [
   scenario s3a (Alternative) ''Customer does not exist
   and has to be created'' [
      step s1 (Actor) ''Selects the action AddCustomer''
      step s2 <extendedBy uc_1_6_CreateCustomer> ]]
step s3 (System) ''Shows the customer's data, namely his
  full name and fiscal identification''
step s4 (Actor) ''Creates one or more invoice lines'' [
  scenario s4a (Nested) ''Create Invoice Lines'' repeat
  ''1.. *''[
     step s1 (Actor) ''Selects the action AddInvoiceLine''
     step s2 (Actor) ''Selects a product and enters the
    quantity of items''
      step s3 (System) ''Updates the InvoiceLine and
    Invoice values'' ] ]
step s5 (Actor) ''Selects the action Confirm''
step s6 (System) ''Validates the submitted data'' [
   scenario s6a (Exception) ''Customer not yet selected''
   [
      step s1 (System) ''Check that a customer was not
      yet selected''
      step s2 (System) ''Displays an error message''
      nextStep s0.s2]
```

```
      scenario s6b (Exception) ''Invoice lines not yet
      added'' [
      step s1 (System) ''Check that there is not at least
    one invoice line added''
      step s2 (System) ''Displays an error message''
      nextStep s4]]
step s7 (System) ''Creates the Invoice information, which
  shall be in the 'Pending' state''
step s8 (System) ''Displays a success message of 'Invoice
  Created and Waiting for Approval'''']
```

UseCase uc_1_3_SendInvoices

```
[...]
mainScenario s0 (Main) [
step s1 (System) ''Shows the interaction space
  SendInvoices, namely with the following settings [...],
  and with available actions: Send, Cancel''
step s2 (Actor) ''Sets criteria to send invoices [...],
  namely shall select just one Customer and the
  respective Invoices to send''
step s3 (Actor) ''Selects the action Send''
step s4 (System) ''Validates the send criteria''
step s5 (System) ''Creates an email message with all
  selected invoices'' [
  scenario s5a (Nested) ''Print and Attach Invoices'' [
    step s1 <include uc_1_4_PrintInvoice> repeat ''for
  each selected invoice''
    step s2 (System) ''Attach the printed PDF files into
  the email message'']]
step s6 (System) ''Send email message to the Customer''
step s7 (System) ''Displays a success message of
  'Invoices Sent to the Customer'''']
```

UseCase uc_1_6_CreateCustomer

```
[...]
mainScenario s0 (Main) ''Happy Flow'' [
step s1 (System) ''Shows the interaction space
  CreateCustomer, with empty fields, and with available
  actions (Confirm, Cancel); by default, the customer's
  creation date is filled with the current date''
step s2 (Actor) ''Fills the customer fields [...]''
step s3 (Actor) ''Select the action Confirm''
step s4 (System) ''Validates the submitted data''
step s5 (System) ''Creates the Customer information''
step s6 (System) ''Shows the previous interaction space'']
```

UseCase uc_4_ReceiveClosedInvoices

```
[...]
mainScenario s0 (Main) ''Happy Flow'' [
step s1 (Event:Timer) ''Starts on 1st of January at 21:00,
  repeat every year''
  event ev4_BeginningOfTheYear
step s2 (System) ''Produces the list of closed invoices
  from e_Invoice''
step s3 (System) ''Sends the list of closed invoices to
  the a_ERP''
step s4 (Actor) ''Receives the list of closed invoices''
step s5 (System) ''Moves the closed invoices from
  e_Invoice to e_ClosedInvoice''
step s6 (System) ''Register success operation in Log'' ]
```

APPENDIX B: SUMMARY OF LINGUISTIC PATTERNS FOR REQUIREMENTS SPECIFICATION: FOCUS ON USE CASES

G. LINGUISTIC PATTERNS

```
Actor

UseCase
  UseCaseRelation
    UCExtendRelation
    UCIncludeRelation
  UseCaseScenario
    UseCaseScenarioStep

DataEntity (not discussed in this paper)
```

H. RECOMMENDED VOCABULARY

```
Actor
  ActorType: User | ExternalSystem

UseCase
  UseCaseType:
    EntityCreate|EntityRead|EntityUpdate|EntityDelete
    [...]
  UseCaseRelation
    UCExtendRelation:
      extensionPoints, extends, onExtensionPoint
    UCIncludeRelation: includes
  UseCaseScenario
    UseCaseScenarioType:
      Main | Nested | Alternative | Exception
   Step
      StepType: Actor | System | Event
      StepUCType: Include | Extend | ExtendedBy
      Rejoin at
```

REFERENCES

[1] K. Pohl, *Requirements Engineering: Fundamentals, Principles, and Techniques*. Berlin, Germany: Springer-Verlag, 2010.

[2] S. Robertson and J. Robertson, *Mastering the Requirements Process*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2006.

[3] J. L. Eveleens and C. Verhoef, "The rise and fall of the chaos report figures," *IEEE Softw.*, vol. 27, no. 1, pp. 30–36, Jan. 2010.

[4] *Chaos Summary 2009 Report, the 10 Laws of Caos*, Standish Group, Boston, MA, USA, 2009.

[5] B. Kovitz, *Practical Software Requirements: Manual of Content and Style*. Shelter Island, NY, USA: Manning, 1998.

[6] S. Withall, *Software Requirements Patterns*. Unterschleißheim, Germany: Microsoft Press, 2007.

[7] J. Verelst, A. R. Silva, H. Mannaert, D. A. Ferreira, and P. Huysmans, "Identifying combinatorial effects in requirements engineering," in *Proc. EEWC*. Cham, Switzerland: Springer, 2013, pp. 88–102.

[8] A. R. Silva, J. Verelst, H. Mannaert, D. A. Ferreira, and P. Huysmans, "Towards a system requirements specification template that minimizes combinatorial effects," in *Proc. 9th Int. Conf. Qual. Inf. Commun. Technol.*, Sep. 2014, pp. 124–129.

[9] D. Fernéndez, S. Wagner, M. Kalinowski, M. Felderer, P. Mafra, A. Vetro, T. Conte, M. T. Christiansson, D. Greer, C. Lassenius, and T. Männistö, "Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice," *Empirical Softw. Eng.*, vol. 22, no. 5, pp. 2298–2338, 2017.

[10] *IEEE Recommended Practice for Software Requirements Specifications*, Standard 830-1998, 1998, pp. 1–40, doi: 10.1109/IEEESTD.1998.88286.

[11] *Guide for Writing Requirements, V.3*, INCOSE, San Diego, CA, USA, 2019.

[12] I. Jacobson, *Object-Oriented Software Engineering—A Use Case Driven Approach*. Reading, MA, USA: Addison-Wesley, 1992.

[13] C. Videira and A. R. Silva, "Patterns and metamodel for a natural-language-based requirements specification language," in *Proc. of CAiSE Short Paper*, Jun. 2005, pp. 1–6.

[14] C. Videira, D. Ferreira, and A. R. Silva, "A linguistic patterns approach for requirements specification," in *Proc. 32nd EUROMICRO Conf. Softw. Eng. Adv. Appl. (EUROMICRO)*, Aug. 2006, pp. 302–309.

[15] A. Cockburn, *Writing Effective Use Cases*. Reading, MA, USA: Addison-Wesley, 2001.

[16] R. Wirfs-Brock and J. Schwartz, "The art of writing use cases," in *Proc. Tutorial OOPSLA Conf.*, 2001, pp. 1–159.

[17] L. Constantine and L. A. Lockwood, "Structure and style in use cases for user interface design," in *Object Modeling and User Interface Design*. Reading, MA, USA: Addison-Wesley, 2001, pp. 245–280.

[18] S. Lilly, "Use case pitfalls: Top 10 problems from real projects using use cases," in *Proc. Technol. Object-Oriented Lang. Syst. (TOOLS)*, Aug. 1999, pp. 174–183.

[19] K. Cox, K. Phalp, and M. Shepperd, "Comparing use case writing guidelines," in *Proc. 7th Int. Workshop Requirements Eng., Found. Softw. Qual.*, 2001, pp. 101–112.

[20] A. Durán, B. Bernárdez, M. Toro, R. Corchuelo, A. Ruiz, and J. Pérez, "Expressing customer requirements using natural language requirements templates and patterns," in *Proc. IMACS/IEEE CSCC*, Jul. 1999, pp. 1–6.

[21] A. R. Silva, "Linguistic patterns and linguistic styles for requirements specification (I): An application case with the rigorous RSL/business-level language," in *Proc. 22nd Eur. Conf. Pattern Lang. Programs*, Jul. 2017, pp. 1–27.

[22] A. R. Silva and D. Savić, "Linguistic patterns and linguistic styles for requirements specification: Focus on data entities," *Appl. Sci.*, vol. 11, no. 9, p. 4119, Apr. 2021, doi: 10.3390/app11094119.

[23] N. Fuchs, K. Kaljurand, and T. Kuhn, "Attempto controlled english for knowledge representation," in *Reasoning Web*. Cham, Switzerland: Springer, 2008, pp. 104–124.
[24] S. Schneider, *The B-Method: An Introduction*. London, U.K.: Palgrave Macmillan, 2001.
[25] T. Kuhn, "A survey and classification of controlled natural languages," *Comput. Linguistics*, vol. 40, no. 1, pp. 121–170, Mar. 2014.
[26] A. R. Silva, "Rigorous specification of use cases with the RSL language," in *Proc. ISD, AIS*, 2019, pp. 1–12.
[27] A. C. Paiva, D. Maciel, and A. R. Silva, "From requirements to automated acceptance tests with the RSL language," in *Communications in Computer and Information Science*, vol. 1172. Cham, Switzerland: Springer, 2020, pp. 39–57.
[28] I. Jacobson, I. Spence, and B. Kerr, "Use-case 2.0," *Commun. ACM*, vol. 59, no. 5, pp. 61–69, 2016.
[29] OMG. (2017). *Unified Modeling Language, Version 2.5.1*. [Online]. Available: http://www.omg.org/spec/UML/
[30] L. Constantine, S. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centred Design*. Reading, MA, USA: Addison-Wesley, 1999.
[31] I. Jacobson, "Use cases-Yesterday, today, and tomorrow," *Softw. Syst. Model.*, vol. 3, no. 3, pp. 210–220, 2004.
[32] D. Savić, S. Vlajić, S. Lazarević, I. Antović, V. Stanojević, M. Milić, and A. R. Silva, "SilabMDD: A use case model-driven approach," in *Proc. ICIST*, Mar. 2015, pp. 1–6.
[33] A. R. Silva, "Use case specification using the SilabReq domain specific language," *Comput. Inform.*, vol. 34, no. 4, pp. 877–910, 2016.
[34] A. R. Silva, J. Saraiva, R. Silva, and C. Martins, "XIS-UML profile for eXtreme modeling interactive systems," in *Proc. 4th Int. Workshop Model-Based Methodolog. Pervas. Embedded Softw. (MOMPES)*, Mar. 2007, pp. 55–66.
[35] A. Ribeiro and A. R. Silva, "XIS-mobile: A DSL for mobile applications," in *Proc. 29th Annu. ACM Symp. Appl. Comput.*, Mar. 2014, pp. 1316–1323.
[36] A. Ribeiro and A. R. Silva, "Evaluation of XIS-mobile, a domain specific language for mobile application development," *J. Softw. Eng. Appl.*, vol. 7, no. 11, pp. 906–919, 2014.
[37] R. Schwitter, "Controlled natural languages for knowledge representation," in *Proc. 23rd Int. Conf. Comput. Linguistics, Posters*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 1113–1121.
[38] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.
[39] D. de Almeida Ferreira and A. R. Silva, "RSL-IL: An interlingua for formally documenting requirements," in *Proc. 3rd Int. Workshop Model-Driven Requirements Eng. (MoDRE)*, Jul. 2013, pp. 40–49.
[40] OMG. (2017).*System Modeling Language, Version 1.5*. [Online]. Available: http://www.omg.org/spec/SysML/
[41] S. Tiwari and A. Gupta, "A systematic literature review of use case specifications research," *Inf. Softw. Technol.*, vol. 67, pp. 128–158, Nov. 2015, doi: 10.1016/j.infsof.2015.06.004.
[42] J. Robertson and S. Robertson, *Volere Requirements Specification Templates*, 2012. [Online]. Available: https://www.volere.com
[43] M. Kamal, M. Ahmed, and M. El-Attar, "Use case-based effort estimation approaches: A comparison criteria," in *Proc. Int. Conf. Softw. Eng. Comput. Syst.* Springer, 2011, pp. 735–754, doi: 10.1007/978-3-642-22203-0_62.
[44] K. Cox and K. Phalp, "Exploiting use case descriptions for specification and design an empirical study," in *Proc. 7th Int. Conf. Empirical Assessment Softw. Eng.*, 2003, pp. 8–10.
[45] M. El-Attar and J. Miller, "A subject-based empirical evaluation of SSUCD's performance in reducing inconsistencies in use case models," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 477–512, Oct. 2009, doi: 10.1007/s10664-008-9101-9.
[46] C. B. Achour, C. Rolland, N. A. M. Maiden, and C. Souveyet, "Guiding use case authoring: Results of an empirical study," in *Proc. IEEE Int. Symp. Requirements Eng.*, Jun. 1999, pp. 36–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=647646.731122
[47] J. C. S. do Prado Leite, G. D. S. Hadad, J. H. Doorn, and G. N. Kaplan, "A scenario construction process," *Requirements Eng.*, vol. 5, no. 1, pp. 38–61, Jul. 2000, doi: 10.1007/PL00010342.
[48] A. H. Dutoit and B. Paech, "Rationale-based use case specification," *Requirements Eng.*, vol. 7, no. 1, pp. 3–19, Apr. 2002, doi: 10.1007/s007660200001.
[49] D. Kulak and E. Guiney, *Use Cases: Requirements in Context*. Reading, MA, USA: Addison-Wesley, 2012.
[50] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *Proc. 17th IEEE Int. Requirements Eng. Conf.*, Aug. 2009, pp. 317–322.
[51] Y. Wautelet, S. Heng, M. Kolp, and I. Mirbel, "Unifying and extending user story models," in *Advanced Information Systems Engineering* (Lecture Notes in Computer Science), vol. 8484. Cham, Switzerland: Springer, 2014, pp. 211–225.
[52] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, and S. Brinkkemper, "Improving agile requirements: The quality user story framework and tool," *Requirements Eng.*, vol. 21, no. 3, pp. 383–403, Sep. 2016, doi: 10.1007/s00766-016-0250-x.
[53] E.-M. Schön, D. Winter, M. J. Escalona, and J. Thomaschewski, "Key challenges in agile requirements engineering," in *Proc. Int. Conf. Agile Softw. Develop. (XP)*. Cham, Switzerland: Springer, 2017, pp. 37–51.
[54] Y. Wautelet, S. Heng, D. Hintea, M. Kolp, and S. Poelmans, "Bridging user story sets with the use case model," in *Advances in Conceptual Modeling* (Lecture Notes in Computer Science), vol. 9975. Cham, Switzerland: Springer, 2016, pp. 127–138, doi: 10.1007/978-3-319-47717-6_11.
[55] M. Elallaoui, K. Nafil, and R. Touahni, "Automatic transformation of user stories into UML use case diagrams using NLP techniques," *Proc. Comput. Sci.*, vol. 130, pp. 42–49, Jan. 2018, doi: 10.1016/j.procs.2018.04.010.
[56] I. K. Raharjana, D. Siahaan, and C. Fatichah, "User stories and natural language processing: A systematic literature review," *IEEE Access*, vol. 9, pp. 53811–53826, 2021.
[57] QRA Corp., "21 top engineering tips: How to write an exceptionally clear requirements document," Halifax, NS, USA, White Paper, 2018.
[58] A. A. Efremov and K. I. Gaydamaka, "INCOSE guide for writing Requirements. Translation experience, adaptation perspectives," in *Proc. CEUR Workshop*, 2019, pp. 164–178.
[59] QRA Corp., "Automating the INCOSE guide for writing requirements," Halifax, NS, USA, White Paper, 2019.
[60] M. Arrabito, A. Fantechi, S. Gnesi, and L. Semini, "An experience with the application of three NLP tools for the analysis of natural language requirements," in *Proc. Int. Conf. Qual. Inf. Commun. Technol.* Cham, Switzerland: Springer, 2020, pp. 488–498.
[61] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proc. SIGCHI Conf. Hum. Factors Comput. Syst. (CHI)*, May 1993, pp. 24–29.

ALBERTO RODRIGUES DA SILVA received a degree in informatics engineering from the New University of Lisbon, in 1989, a M.S. degree in electrical and computer engineering, a Ph.D. degree in computer science and engineering, and a Habilitation degree in computer science and engineering from the Technical University of Lisbon, in 1992, 1999, and 2016, respectively.

He is currently Associate Professor (Habilitation) with the Instituto Superior Técnico, Universidade de Lisboa (IST-UL), and Senior Researcher with INESC-ID Lisboa. He has taught more than 4000 students and supervised more than 10 Ph.D. students and 70 M.Sc. students. He has authored 6 technical books and more than 200 publications in journals, conferences, and workshops with peer review, and has been also editor of 5 scientific books. His research interests include information systems, software engineering, model-driven engineering, requirements engineering, document automation, and project management, and their application in multiple domains.

Dr. Silva is a member of the OE, (Portuguese Chartered Engineers Association), and the PMI (Project Management Institute). He is a Senior Member of the Association of Computer Machinery (ACM). He received awards and professional certifications such as JEEP, Scrum Master, PMP, and IST Excellent Teaching.