RSL-PL: A Linguistic Pattern Language for Documenting Software Requirements

David de Almeida Ferreira, Alberto Rodrigues da Silva
INESC-ID, Instituto Superior Técnico (IST), Lisbon, Portugal
{david.ferreira, alberto.silva}@inesc-id.pt

***Abstract*—Software requirements are traditionally documented in natural language (NL). However, despite being easy to understand and having high expressivity, this approach often leads to well-known requirements quality problems. In turn, dealing with these problems warrants a significant amount of human effort, causing requirements development activities to be error-prone and time-consuming. This paper introduces RSL-PL, a language that enables the definition of linguistic patterns typically found in well-formed individual NL requirements, according to the field's best practices. The linguistic features encoded within RSL-PL patterns enable the usage of information extraction techniques to automatically perform the linguistic analysis of NL requirements. Thus, in this paper we argue that RSL-PL can improve the quality of requirements specifications, as well as the productivity of requirements engineers, by mitigating the continuous effort that is often required to ensure requirements quality criteria, such as clearness, consistency, and completeness.**

***Index Terms*—Requirements Engineering, Linguistic Analysis, Requirements Linguistic Patterns, Information Extraction.**

I. Introduction

To deliver successful software, one must document requirements in a *clear* and *precise* manner, as poorly defined requirements often lead to well-known requirement defects [1]. However, despite some quality problems due to the inherent characteristics of natural language (NL) such as ambiguity, the majority of requirements documents are still documented with such Requirements Specification Language (RSL) [2]. This is mainly due to the expressive power of NL, and also because business stakeholders require *no training* for being able to communicate with NL. Being the "least common denominator" between business stakeholders and the development team, NL supports an *effective communication* of requirements.

Balancing the tradeoffs of using NL to document requirements, namely perfecting such requirements specifications and relying on the typical multi-representation strategy based on complementary conceptual models to mitigate the negative effects of ambiguity, often entails a significant amount of human effort [3]. For instance, enforcing the consistency between NL and the respective conceptual models significantly increases the burden on requirements engineers.

Moreover, there is a clear lack of tool support for the *linguistic analysis* of requirements documented in NL, which is still manually performed by requirements engineers. Such analysis is crucial to deal with requirements at the semantic level of their NL representations, instead of following the traditional tool support approach of only handling requirements metadata (i.e., requirements attributes and relations). Although current Natural Language Processing (NLP) techniques are still unable to fully "understand" NL text in a generalized manner [4], there are some simplifications that can improve the current tool support for the linguistic analysis of NL requirements. Such improvement would significantly lessen the burden on requirements engineers to enforce requirements clearness, consistency, and completeness.

To address typical requirements development problems [3], both in terms of productivity and quality, we advocate a requirements specification approach based on linguistic patterns, such as RSLingo [5]. This approach follows a *multi-language* strategy to support a requirements formalization process based on linguistic patterns that enables one to document in a precise manner requirements originally stated in NL.

This paper introduces RSL-PL, one of RSLingo's specific languages. RSL-PL is a pattern definition language based on linguistic patterns that abstract over well-formed requirements representations, according to the best practices of Requirements Engineering (RE) [2]. The purpose of RSL-PL is to define linguistic patterns that enable one to deal with the linguistic concerns of the textual representations of NL requirements, so that such patterns can be used by RSLingo's toolset during the information extraction process. This paper emphasizes the usefulness of RSL-PL for the automation of the requirements linguistic analysis within this process, by presenting its constructs, namely the *elements* that define lexical constraints upon the token types that can be matched against each position of a pattern, and the *operators* that combine them into more complex linguistic patterns.

The remainder of this paper is structured as follows. Section II presents the motivation for RSL-PL. Section III states the design principles of RSL-PL and defines its constructs. Section IV presents a set of guidelines for defining linguistic patterns that, through the examples, illustrates the practical feasibility of RSL-PL. Finally, Section V concludes this paper.

II. Background

Considering the premises introduced in the previous section, regarding the common practices of the requirements documentation activity, we consider that the quality of requirements specification still strongly depends on the expertise of whoever is performing this activity. Given that most RE activities are still manually performed and involve a large amount of information, to produce a high quality requirements specification these activities warrant an experienced requirements engineer

978-1-4799-0948-3/13/$31.00 © 2013 IEEE

with a vast skills set. Thus, to avoid large discrepancies in the outcome of the RE process (i.e., the requirements baseline), we advocate that the *quality* of requirements specifications and the *productivity* of the requirements documentation activity can be increased through the *formalization* of requirements. However, since requirements are typically documented in NL, we consider that this requirements formalization process must entail the automation of the *linguistic analysis* of the textual NL representation of such requirements.

A. The RSLingo Approach

RSLingo is an information extraction approach based on linguistic patterns, which follows a multi-language strategy based on two languages [5]: RSL-PL (Pattern Language) for defining linguistic patterns, and RSL-IL (Intermediate Language) that acts as a formal requirements *interlingua* [6]. The main goals of RSLingo are: (1) to perform the linguistic analysis; and (2) to formalize requirements originally stated in NL by addressing them at a deeper semantic level. To attain these goals, RSLingo is based on lightweight NLP techniques. After enriching the surface requirements text with the required linguistic features, RSLingo follows an information extraction mechanism based on a pattern alignment algorithm [7] that can be used to exploit the syntactic–semantical alignment of words within a NL sentence. This technique allows to capture requirements-related information with the linguistic patterns defined in RSL-PL. In turn, through previously declared transformations between RSL-PL and RSL-IL, one is able to automatically generate semantically equivalent (but formal) RSL-IL specifications. Since the obtained specifications are amenable to be processed by computers, they enable the automatic verification of some requirements quality criteria, such as *clearness*, *consistency*, and *completeness*. The requirements formalization process supported by RSLingo, as well as the part played by RSL-PL, are depicted in Figure 1.

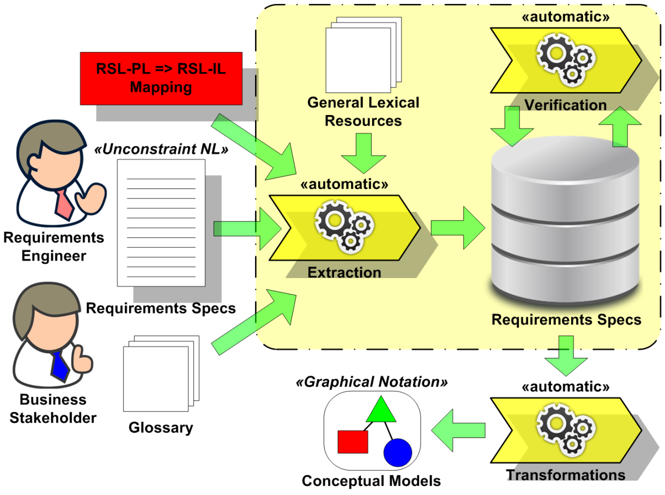

Figure 1. Overview of the RSLingo approach.

It is noteworthy that such a decoupling of linguistic concerns from requirements formalization issues – attained with this multi-language strategy – addresses the previously mentioned requirements specification problems without introducing new and disruptive languages that business stakeholders must learn (which, in turn, could hinder them from directly authoring and validating their own requirements). RSLingo is meant to enable the active participation of business stakeholders in the requirements development workflow of the RE process, by enabling them to directly author their own requirements in NL. Moreover, since this requirements formalization process can be automated, the feedback on requirements quality problems is less reliant on the expertise of requirements engineers.

B. Natural Language Documentation

As technical documentation, requirements specifications written in NL should follow the best practices of *technical writing* [8]. The goal of following these best practices is improving the *readability* of requirements specifications, as well their *understandability*, in order to support an *effective communication* between different stakeholder. These best practices are often described in terms of *wording* and *writing style* guidelines for creating NL sentences that represent individual requirements. The literature on RE (specially textbooks [2], [8]) contains several recommendations regarding the writing styles that requirements engineers (as technical writers) should adhere to improve the quality of NL requirements specifications. These recommendations focus mostly on syntactic structures that are regarded as being well-formed for documentation purposes. As illustrated in Listing 1, these syntactic structures work as templates distilled from experience.

Listing 1. Example of requirements templates extracted from the literature.

Patterns for individual requirements:

`<when> THE SYSTEM SHALL <process> <object>.`
”*R114: When the glass break detector detects the damaging of a window, the system shall inform the head office of the security service.*” (adapted from [2])

`THE SYSTEM SHALL <action> <to_whom> <what>.`
”*The system shall ask the operator for the passenger name.*” (adapted from [3])

`<actor> <action> <what> <how>.`
”*A user can view landscape files in AutoCAD format.*” (adapted from [8])

`Each <owner_entity_name> shall have an unique ID (...).`
”*Each loan approval decision rule shall have an ID by which it can be referred.*” (adapted from [9])

Patterns for Use Case steps:

`<subject> <verb> <direct_object> <prepositional_phrase>.`
”*The system deducts the amount from the account balance.*”
(adapted from [10])

Patterns for atomic business rules:

`If <condition> then <action>.`
”*If repeated customer requests discount, then offer 5% discount for interior window cleaning line items.*” (extracted from [11])

`On <event> if <condition> then <action>.`
”*On close-of-week if out-standing payment then issue reminder letter.*”
(extracted from [11])

Recognizing that these requirements templates are by themselves project artifacts that play a key role regarding the quality assurance of requirements specifications, following the best practice of *reusing knowledge* from previous projects, we consider of the utmost importance to collect and organize these patterns recurring in requirements specifications because they are the result of a successive refinement of knowledge accumulated by requirements engineers.

As presented in Listing 1, most of the approaches to define requirements templates are based on syntactic structures. These structures are created by keeping *function words* [12], whereas *content words* [12] are replaced by placeholders that abstract them [2]. To help requirements engineers to instantiate these templates, each placeholder contain a description that provides a hint on the expected content [2]. But, following such an approach warrants human cognition to determine which words can fill each placeholder. Thus, it is unsuitable to be used by automated information extraction techniques.

In addition to these *knowledge reuse* concerns, which are often realized through catalogues of well-defined requirement patterns [9], these templates should support the *linguistic analysis* of NL requirements. This technique is crucial to improve the understanding and communication of requirements. The guidelines provided by the literature on RE often recommend the identification of lexical words, such as *nouns* and *verbs* [13]. Although there are some tools that are able to (partially) support this technique, most of the times the linguistic analysis of textual requirements specifications is manually performed by requirements engineers.

To support the linguistic analysis of requirements in an automated manner, the best approach is to rely on well-known NLP techniques. For instance, bearing in mind the state of the art in terms of lightweight NLP techniques, one could argue that *Chunking* [4] could solve the problem. However, due to its roots on regular expressions [14], this technique is inadequate for dealing with the flexibility of NL: it would yield a boolean outcome (i.e., it is either a match or not). Despite being technically possible to address the problems that led to the development of RSL-PL with an approach based on regular expressions, it would require very complex regular expressions (or, alternatively, a large number of smaller but very similar regular expressions) to deal with partial matches or unexpected tokens. Therefore, such approach is not feasible in practice because it would be extremely difficult to manage. An information extraction approach (such as RSLingo) requires the robustness and flexibility of fuzzy matching during the detection of pattern alignments [7]. For instance, Listing 2 illustrates the limitations of a *RegexpChunkRule* [4] when parsing a requirement sentence in NL.

Listing 2 identifies noun phrases (i.e., chunks) consisting of an optional determiner followed by one or more nouns. The chunk parser correctly matches three noun phrases (lines 8, 11, and 14). However, this approach requires one to explicitly take into account the possibility that requirements textual representations contain additional tokens between the elements defined by the requirements pattern; otherwise the pattern will not match as it was intended, if unexpected tokens are encountered. For example, the last noun phrase was not correctly identified (lines 16–18) because the rule does not take into account noun modifiers (in this case "given") based on the past-participle construction (line 17).

Listing 2. Limitations of Chunking based on regular expressions.

```
RegexpChunkRule: ’NP:{<DET>?<N.∗>+}’

Requirement: ”The system shall allow the call center operator to list all bank
accounts of a given client.”

Output:
(S
  (NP The/DET system/N)
  shall/MOD
  allow/V
  (NP the/DET call/N center/N operator/N)
  to/TO
  list/V
  (NP all/DET bank/N accounts/N)
  of/P
  a/DET
  given/VN
  (NP client/N)
  ./.)
```

Moreover, these rules only support the definition of constraints based on the part-of-speech of each token, disregarding other relevant restrictions such as the type of the token. Also, when dealing with words (one of the possible token types), these rules do not consider lexical and semantical relations, such as those provided by WordNet [15] (e.g., the notion of *synset*). Furthermore, while defining requirements templates it is desirable to have the ability to specify some pivotal words in verbatim (e.g., function words). However, since chunk rules are regular expressions defined in terms of word's part-of-speech tags (i.e., a single level of abstraction), it does not support the creation of such fixed anchors within a linguistic pattern.

Therefore, the RSLingo's specific information extraction needs are not supported (to the best of our knowledge) by any other state of the art techniques and languages.

III. Linguistic Patterns with RSL-PL

To address the problems presented in the previous section, we developed RSL-PL as a concern-specific language for defining *linguistic patterns*[1] aligned with the recommendations and style guidelines to write requirements in NL, which provide linguistic-oriented abstractions of the requirements templates mentioned above [2]. However, RSL-PL goes further, as it deals with NL tokens at different abstraction levels based on their linguistic features, without disregarding aspects related with its practical usability and maintainability of pattern catalogues. RSL-PL provides the foundations for the extensibility of linguistic patterns, as well as for the (re)use of catalogues of well-formed requirements templates.

Since RSL-PL supports the representation of a wide range of linguist constructions in terms of lexicon and syntax, it provides a sort of *extensibility mechanism* in the sense that

[1] In the context of RSLingo, we refer to requirements templates as *linguistic patterns* because, for coping with its information extraction specific needs, they must convey more than just syntactic-related restrictions.

it enables the definition of new linguistic patterns to cope with the writing styles and idiosyncrasies of different projects and domains. Thus, the flexibility of RSL-PL in supporting new or tailored linguistic patterns significantly enlarges the coverage in terms of NL requirements representations that are recognized by the RSLingo approach.

Moreover, RSL-PL addresses the automation problem by enriching the placeholders provided by typical requirements templates with computer-processable constraints on the NL tokens (e.g., words) that can fill each of those placeholder positions. In addition to enabling the synthesis of well-formed requirements templates, RSL-PL goes further by allowing the establishment of mappings to RSL-IL, the language that allows the formal representation of requirements within the RSLingo approach. It should be noted that RSL-IL provides a low-level language that acts as a *mold to be filled* with information extracted from NL requirements through their alignment with the linguistic patterns defined in RSL-PL.

A. Design Principles

After discussing the motivation for creating a new language such as RSL-PL, the following paragraphs present the most relevant design principles that were taken in consideration while developing it, which are somewhat expressed by the semantics and notation of its constructs.

a) ***Domain-independency****:* RSLingo follows a linguistic--oriented perspective to the extraction of relevant information from NL requirements representations. Therefore, since there is a strong alignment with NL, RSL-PL inherits its *universality* characteristic, in the sense that it is general enough to define linguistic patterns for different application domains.

b) ***Cognitive-alignment****:* RSL-PL constructs reflect the linguistic foundations of RSLingo by enabling the definition of linguistic patterns with elementary notions of NL grammar. For instance, there are just a few basic token types (e.g., word, symbol), which can be refined considering a limited set of lexical properties and relations (e.g., part-of-speech and synonymy, respectively), or simply by providing a specific token in verbatim. Additionally, the concept of *linguistic pattern* is intrinsically related with the notion of NL sentence. Thus, a RSL-PL pattern intuitively maps to an *ordered sequence* of NL tokens whose scope is entailed within the boundaries of a common sentence written in plain English.

c) ***Simplicity****:* Coping with *all* and *every* aspect of NL with today's technology is a difficult (if not impossible) task to accomplish. Within these current limitations, for attaining a pragmatic information extraction approach for requirements specifications written in NL, we followed the guideline of not introducing unnecessary language elements when creating a specialized language [16].

d) ***Usability****:* For mitigating the common adoption resistance of new languages, the RSL-PL constructs were designed to be moderately easy to understand by readers and straightforwardly applied by writers. The concrete syntax of RSL-PL was designed in such a manner that it does not require a sophisticated tool to be edited: a simple text editor suffices for reading and writing the RSL-PL, even without syntax highlighting and autocompletion.

e) ***Computability****:* For attaining high quality requirements specifications in a cost-effective manner, the approach supported by RSL-PL strongly relies on the automation of the typical manual process of analyzing and translating informal into formal requirements specifications. Given this dependence on the ability to automate the linguist analysis of requirements specifications (written in NL), during the design of the RSL-PL constructs we carefully considered technological issues regarding the parsing and further processing of the linguistic patterns defined with them.

f) ***Maintainability****:* Thinking both in terms of *extensibility* and *reusability*, we took special care regarding how RSL-PL patterns can be maintained. On one hand, it is important to support customization of existing patterns in order for them to be applied in new application domains. On the other hand, it is important that the growth of the catalogue of recognized linguistic pattern is performed with some kind of tool support, otherwise the ad-hoc expansion of this resource might lead to redundant linguistic patterns.

B. Overview of RSL-PL Design

RSL-PL's *abstract syntax* is defined by the metamodel depicted in Figure 2, which highlights the main notions and relations underlying its constructs. Since we favored *simplicity* as the most important design principle, an RSL-PL construct is either an *atomic element*, or an *operator* that combines these atomic `Elements` into more complex structures (e.g., nested `Patterns`). RSL-PL provides *four* atomic `Elements`, namely: (1) `Word`, to represent the occurrence of a typical dictionary word; (2) `DataType`, to support the recognition of primitive data types (e.g., an integer); (3) `Symbol`, to express a single occurrence of punctuation marks or any another character, except for white spaces (that are simply ignored) and alphanumeric characters (to avoid overlap with `Words` or `DataType` tokens); and (4) `Reference`, to provide a placeholder for both the representation of the requirement's *unique identifier*, and the insertion of *trace links* to other requirements or external artifacts.

Additionally, to compose these atomic elements, RSL-PL provides *two* operators: (1) `Concatenation`, which enables the creation of `Patterns` by chaining together these atomic `Elements`; and (2) `Composition`, which enable `Patterns` to be interchangeably treated themselves as `Elements` when the `Concatenation` operator is being applied to form new `Patterns`, thus supporting the creation of *nested structures*. Also, to increase the expressive power of RSL-PL, the `Composition` operator provides the basic means to support the quantification of `Patterns` included by other `Patterns`, in the sense that it allows to specify how often a given sub-`Pattern` is allowed to occur within the context of the including `Pattern`. Within these more complex structures, each `Pattern` abstracts over a constituent with a specific syntactical function (or that plays a

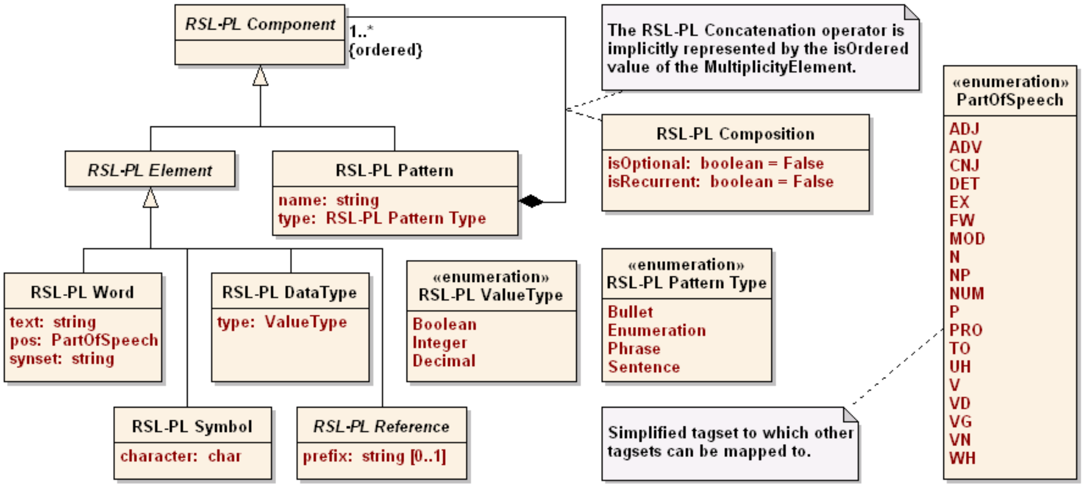

Figure 2. The RSL-PL metamodel.

semantic role[2]) regarding the sentence that the outer `Pattern` represents.

C. Description of RSL-PL Constructs

The semantics of each of the RSL-PL constructs mentioned above is explained in the following paragraphs. Also, the accompanying examples unveil their *concrete syntax*.

*g) **Word Element:*** The `Word` element represents any of the *three types* of compound words, namely *closed compounds*, *hyphenated compounds*, and *open compounds*. However, to identify a *open compound* (i.e., two or more words separated by white spaces) as a single `Word`, the respective NL text must explicitly indicate that these words must be treated as a single unit of meaning, otherwise they are treated separately.

Regarding its concrete syntax, a `Word` element is represented by a single capital '`W`' letter. Figure 3, illustrates the *alignment* of a linguistic pattern (the upper sequence) with the beginning of a traditional "shall" requirements statement.

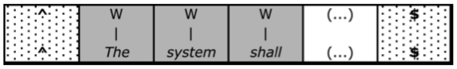

Figure 3. Example of the concrete syntax of an RSL-PL `Word` element.

However, as illustrated in Figure 3, a pattern consisting of three '`W`' would match[3] any sequence of three consecutive dictionary words; that is, in its simplest form, a `Word` element would not be more useful than defining a regular expression for matching such words. Thus, besides the first classification level – whose purpose is to determine whether a token (i.e., a sequence of characters) is classified as being a `Word` based solely on its form –, the RSL-PL language must provide additional mechanisms to restrict the range of possible matches.

To enable the automatic extraction of information at a semantic level, the `Word` element must be parameterized with *semantic selectors* to further refine the range of allowed matches based on linguistic information. This refinement can range from an abstract restriction of the word's lexical category (i.e., its part-of-speech tag) to concretely specifying the word verbatim (i.e., the exact word to be used). In addition to lexical refinements, the `Word` element can be parameterized with semantical restrictions based upon WordNet synsets, namely by exploiting the relations defined among them, and also taking advantage of the explicitly provided definition of their meaning. In short, these parameters constrain the expected token for a given position within a RSL-PL pattern by refining the kinds of words, or even the exact words, that can be matched against that specific `Word` element. For instance, Figure 4 illustrates some of the parameters that are supported by the RSL-PL `Word` element. The fist `Word` only allows *determiners* (i.e., the '`DET`' part-of-speech tag), such as the words "a" or "the". Note that the background of the third column is dark-gray to emphasize that a mismatch occurred, because the second position of the RSL-PL pattern was defined to only allow an *adjective* (i.e., the '`ADJ`' part-of-speech tag), and the corresponding position in the NL sentence is filled by a *noun* (i.e., the '`N`' part-of-speech tag) instead. This problem does not occur with the third `Word` (i.e., the fourth column) because both the RSL-PL element and the NL word share the same part-of-speech tag (i.e., the '`N`' part-of-speech tag). Finally, the fourth `Word` explicitly defines that it will only match with that exact word (i.e., the modal verb "shall").

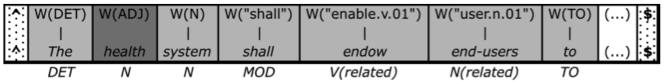

Figure 4. Example of `Word` elements with parameters.

[2] *VerbNet: A Class-Based Verb Lexicon*, <http://verbs.colorado.edu/~mpalmer/projects/verbnet.html> (accessed on April 29th, 2013)

[3] Each match is represented by a light-gray column connecting the `Word` element of the RSL-PL pattern with the respective token of the NL sentence.

*h) **DataType Element**:* Often NL requirements include tokens that represent datatypes typically found in programming languages and database schemas. Additionally, quality attributes (a subset of non-functional requirements) further constrain the manner by which the functionality (described by functional requirements) should be provided by the software system. Such constraints are frequently described in terms of requirements metrics and their respective units [8]. Therefore, in addition to introducing a new classifier for better distinguishing NL tokens, the `DataType` element provides a sort of hookup in RSL-PL that allows the recognition of common datatype values, such as boolean, integer, float (which subsumes decimal), or even strings. When representing the metrics of quality attributes, these datatype values might also include the unit of the metric they represent (e.g., "x transactions/sec" or "x % of incidents leading to failures").

*i) **Symbol Element**:* The RSL-PL `Symbol` element provides the means to represent single-character tokens (e.g., comma, colon, semicolon) that often play an important role within linguistic patterns, namely to establish the relation between the *main clause* of a sentence with the other sentence's constituents (e.g., prepositional phrases, subordinate clauses) [12]. These tokens enable the definition of more complex sentence structures, while reducing its ambiguity by establishing boundaries between these sentence's constituents. Thus, `Symbols` act as *anchors* to better identify subsequences within RSL-PL linguistic pattern, by reducing the number of possible interpretations. Besides reducing ambiguity, these `Symbols` are essential to identify the boundaries of consecutive sentences within a paragraph (i.e., a well-formed sentence must be terminated with a proper punctuation mark).

Regarding its concrete syntax, as illustrated in Figure 5 a `Symbol` element is represented by a single capital '`S`' letter.

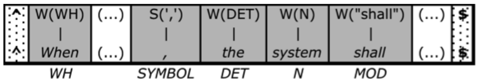

Figure 5. Example of the concrete syntax of an RSL-PL `Symbol` element.

*j) **Reference Element**:* Enabling requirements traceability is essential to better support the activities of the RE process. To enable the identification of implicit knowledge, namely less obvious relations between requirements (e.g., determining all requirements in which a given business entity is acted upon), it is important to support the field's best practices that are based on explicit trace information, either within the requirements textual representations, or through their metadata (i.e., requirement attributes) [17]. These traces between requirements within the same requirements specification (or even between requirements and external documents) are often defined through references. In its essence, a *reference* is a special NL token that allows the reader to uniquely identify another well-defined piece of information, usually another snippet of text (e.g., a paragraph, record, or modeling element), or the whole artifact (e.g., document, database, or model, respectively). The most common approach to represent such references resorts to the target requirement's *unique identifier*, which can be supplemented by a descriptor similar to a citation of a bibliographic reference. This approach copes with application scenarios in which the requirement being referenced is not contained within the same requirements specification.

The concrete syntax chosen to represent a RSL-PL `Reference` element is '`REF`'. Employing a `Reference` within a given position of a RSL-PL `Pattern` implies that it will only match with a sequence of characters (without white spaces) that obey to the following schema: *<req_type><separator><hierarchical_numbering>* (e.g., "FR-1.2.10"). The first part, the requirement type, only allows *uppercase letters*, whereas the last part only allows *digits* and *dots* to create an hierarchical sequence of natural numbers. To ensure that the context of requirements is preserved, the prefix of this hierarchical sequence (i.e., the leftmost $n - 1$ numerical sequences) should be aligned with the requirements specification structure layout.

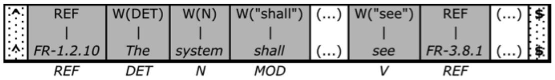

Figure 6. The concrete syntax of an RSL-PL `Reference` element.

*k) **Pattern**:* The previously introduced `Elements` are atomic (i.e., they directly map to a single NL token) and, consequently, are not sufficient by themselves to support an information extraction approach because they only represent an *individual position* within a linguistic pattern. Thus, to represent linguistic patterns in RSL-PL we require the (non--atomic) `Pattern` element, which enables the creation of arrangements of RSL-PL `Elements`, such as the sequence of `Word` elements depicted in Figure 3. To create one of these complex structures according to the metamodel of RSL-PL depicted in Figure 2, this language provides two operators, which are expressed through the composition relationship established between a `Pattern` and its `Components`. The semantics of these two operators is as follows.

*l) **Concatenation Operator**:* The most elementary operator to assemble atomic `Elements` together into a `Pattern` is the `Concatenation` operator. Note that this operator is *implicitly* represented by the composition relationship between a `Pattern` and its `Components`. According to the RSL-PL metamodel, the `Component` values in any given instantiation are *ordered*. This implies that their values at instantiation are *sequentially ordered*. Such mapping is important to support the definition of declarative transformations that use RSL-PL as a source language (e.g., RSL-PL => RSL-IL mapping) [5].

Despite covering a wide range of practical scenarios, by itself the `Concatenation` operator does not support the definition of *recurring* sub-`Patterns` (i.e., specifying that a sub-`Pattern` may appear zero or more times). This operator requires that every occurrence of a given `Component` within a Pattern to be explicitly defined. For instance, to match an arbitrary NL sentence of at most n Word elements, one would need to define a Pattern with exactly n Word elements, in order to ensure that a complete alignment would be possible.

*m) **Composition Operator**:* The purpose of this operator is to allow the definition of sub-Patterns (i.e., well-defined groups that form nested structures) within a larger Pattern. Although the Concatenation operator is simpler than the Composition operator (in the sense that it only deals with flat structures of atomic Elements), these two operators should be regarded as *complementary*. For instance, without the Composition operator, the Concatenation operator would only be able to create sequences of atomic RSL-PL Elements. Only after that can we indifferently intertwine the atomic Elements with these created Patterns. Figure 7 illustrates how Patterns are combined together with the Concatenation and Composition operators.

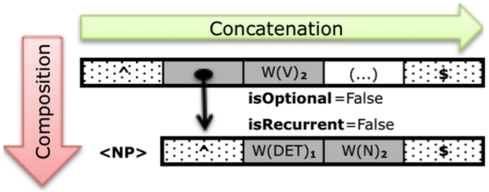

Figure 7. Example of the RSL-PL operators being applied.

The sub-Patterns defined through the Composition operator lay the foundations for improved maintainability, and also to support features similar to those provided by regular expressions. On one hand, sub-Patterns provide a better understanding of the internal grammatical structure of the containing Pattern. This improved understanding helps during the maintenance of the accumulated knowledge, which is realized as the catalogue of RSL-PL Patterns. On the other hand, sub-Patterns provide the basis for mimicking the matching behavior of regular expression quantifiers [14]. This can be achieved through the attributes isOptional and isRecurrent, which respectively specify whether the Pattern is *optional*, and also whether it represents a set of consecutive (unbounded) *repetitions* of the Components sequence that it entails. These two simple attributes support the three most important quantifiers of regular expression [14]: (1) Kleene star, (2) Optional; and (3) Kleene plus.

After defining the semantics of the RSL-PL operators that allow the creation of Patterns, it is important to describe their two attributes: name and type. On one hand, the name attribute, provides the description of the sub-Pattern's role within the containing Pattern. Providing a meaningful name can be helpful for the maintenance of linguistic patterns (e.g., recording the rationale for that sub-Pattern) and to better assist the definition of transformations, namely to use the name of a given sub-Pattern to refer to the capture's value within the target language, instead of using numerical indexes of the positions within the sub-Pattern. On the other hand, the type attribute provides the means to support different sentence-based format of requirements.

IV. Linguistic Pattern Definition Guidelines

RSL-PL was designed to be general enough so that it can *cope with virtually any kind of writing style*. Thus, we provide guidance for creating meaningful and well-formed patterns.

*n) **Guideline for Wording Style**:* There are *three* different styles to write a requirement statement, namely [8]: (1) in the present tense; (2) using modal verbs (i.e., the traditional "shall" statements); and (3) in the imperative mood. Although the outcome of applying the imperative mood might not resemble typical a "requirement statement", it straightforwardly conveys the essence of a requirement. This writing style is commonly used to document the goals of the system-of--interest, and can be directly adapted to use in diagrams, tables, and database-supported requirements repositories. Listing 3 illustrates the differences between these three requirement wording styles, and how can they be defined in RSL-PL.

Listing 3. Wording of an individual requirement's sentence.

```
Present tense: ”A manager generates the status report.”
[?<actor>W(DET) W(N)+ ] [?<action>W(V) ] [?<entity>W(DET) W(N)+ ]

Modal verb: ”A user shall be able to generate the status report.”
[?<actor>W(DET) W(N)+ ] ”shall” ”be” ”able” ”to” [?<action>W(V) ]
        [?<entity>W(DET) W(N)+ ]

Imperative mood: ”Generate the status report.”
[?<action>W(V) ] [?<entity>W(DET) W(N)+ ]
```

*o) **Guideline for Legal Obligation**:* Despite being correlated with the metadata of individual requirements such as the "criticality" (e.g., critical, standard, optional) and/or "prioritization" (e.g., a numeric scale) attributes, requirements engineers often explicitly emphasize the degree of contractual obligation through the wording of the requirement statement [2]. As it was introduced above, different modal verbs can be used to express different legal bindings [18].

According to the field's common practice [2], modal verbs such as *shall*, *should*, and *will* are used to express different degrees of requirements priority and obligation on each of the system's features. Thus, the user must take this convention into account when defining RSL-PL Patterns, to ensure a proper wording according to the expected legal obligations to be captured by those Patterns, as illustrated in Listing 4.

Listing 4. Using modal verbs to express legal obligation.

```
Mandatory: ”A user shall be able to generate the status report.”
[?<actor>W(DET) W(N)+ ] ”shall” ”be” ”able” ”to” [?<action>W(V) ]
        [?<entity>W(DET) W(N)+ ]

Recommended: ”A user should be able to generate the status report.”
[?<actor>W(DET) W(N)+ ] ”should” ”be” ”able” ”to” [?<action>W(V) ]
        [?<entity>W(DET) W(N)+ ]

Desirable: ”A user will be able to generate the status report.”
[?<actor>W(DET) W(N)+ ] ”will” ”be” ”able” ”to” [?<action>W(V) ]
        [?<entity>W(DET) W(N)+ ]
```

*p) **Guideline for Term Definitions**:* As illustrated in Listing 5, this sort of sentences make assertions about the problem space. For instance, they can be used to assign a precise meaning to concepts or auxiliary terms that are used in the requirements specification. Since definitions don't have a truth value (i.e., whether it is satisfied or not), they support the definition of detailed requirements and high-level design from requirements in an arguably correct manner.

Listing 5. `Patterns` for term definitions.

```
Pattern:
[?<term>W(DET) W(N)+ ] ”is” [?<new_term>W(DET) W(N)+ ] [?<condition>
        ”when” [?<entity>W(DET) W(N)+ ] ”is” [?<state> (...) ]]

Domain definition:
[ [ A bank client ] [ is said to be ] [ a ”premium bank client” ] ] [ when [ the total
        amount ] [ of all of his/her bank accounts ] is higher than $ 5.000,00 ].
```

*q) **Guideline for Domain Descriptions**:* The knowledge about the problem space is often made explicit through *descriptive statements*. The described properties are based on natural laws or physical constraints, thus they can be regarded as *domain invariants*. As depicted in Listing 6, the most suitable writing style is the present indicative.

Listing 6. `Patterns` for domain descriptions.

```
Pattern:
[?<entity>W(DET) W(N)+ ] { ”is” | ”has” } [?<entity>W(DET) W(N)+ ]

Domain descriptions:
[ A bank client ] [ is ] [ a user ].
[ A bank client ] [ has ] [ [ one or more ] bank accounts ].
```

*r) **Guideline for User Interactions**:* Besides providing the means to abstract over traditional requirement “shall” statements, RSL-PL is versatile enough to define simpler linguistic patterns that are frequently used in the steps of Use Case descriptions [10]. The present indicative (i.e., the *present tense* wording style) is suitable to describe the interactions between the actors and the system-of-interest. These sentences follow a simple grammar based on the Subject–Verb–Object (SVO) structure, optionally ending with a prepositional phrase.

Listing 7 presents a `Pattern` defined in RSL-PL that is generic enough to match any of the following Use Case steps. However, one might need to enforce a clear distinction between the action verbs available for different actors (e.g., system versus user). In such cases, the best approach is to define different `Patterns` to cope with specific subjects, and thus control the allowed range of action verbs.

Listing 7. `Patterns` for Use Case description steps.

```
Pattern:
[?<actor>W(DET) W(N)+ ] [?<action> W(V) ] [?<entity>W(DET) W(N)+ ]
        [?<prepositional> W(P) [?<related_entity>W(DET) W(N)+ ] ]?
Use Case steps:
1. [ The call center operator ] [ enters ] [ the bank account number ] [ of the bank
        client ].
2. [ The system ] [ displays ] [ the balance ] [ of the bank account ].
3. [ The call center operator ] [ informs ] [ the bank client ] [ of the balance of
        his/her bank account ].
```

V. Conclusion

This paper presents RSL-PL, the language that deals with the linguistic concerns that must be considered when documenting requirements in NL, and which addresses the unique linguistic features required to support an information extraction approach [5]. RSL-PL encodes these concerns as linguistic patterns that abstract over individual requirements considered as being well-formed according to the field's best practices. The flexibility of RSL-PL in defining new linguistic patterns and coping with different requirements writing styles fosters the extensibility of a catalog of linguistic patterns, allowing it to be adapted and reused in different application scenarios.

Acknowledgment

This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under the PEst-OE/EEI/LA0021/2013 project.

References

[1] A. M. Davis, *Software Requirements: Objects, Functions and States*, 2nd ed. Prentice Hall, March 1993.

[2] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam*. Rocky Nook, April 2011.

[3] A. M. Davis, *Just Enough Requirements Management: Where Software Development Meets Marketing*. Dorset House Publishing, May 2005.

[4] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. O'Reilly Media, June 2009, ISBN-13: 978-0596516499.

[5] D. Ferreira and A. Silva, “RSLingo: An Information Extraction Approach toward Formal Requirements Specifications,” in *Proc. of the 2nd Int. Workshop on Model-Driven Requirements Engineering (MoDRE 2012)*. IEEE Computer Society, September 2012, pp. 39–48, ISBN-13: 978-1-4673-4388-6.

[6] ——, “RSL-IL: An Interlingua for Documenting and Reusing Software Requirements,” in *Proc. of the 3rd Int. Workshop on Model-Driven Requirements Engineering (MoDRE 2013)*. IEEE Computer Society, July 2013.

[7] T. Smith and M. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, March 1981.

[8] B. Kovitz, *Practical Software Requirements: Manual of Content and Style*. Manning, July 1998.

[9] S. Withall, *Software Requirement Patterns (Best Practices)*. Microsoft Press, June 2007.

[10] A. Cockburn, *Writing Effective Use Cases*, 1st ed. Addison Wesley, October 2000, ISBN-13: 978-0201702255.

[11] E. Gottesdiener, *The Software Requirements Memory Jogger: A Desktop Guide to Help Soft. and Business Teams Develop and Manage Requirements*. GOAL/QPC, November 2009.

[12] A. DeCapua, *Grammar for Teachers: A Guide to American English for Native and Non-Native Speakers*, 1st ed. Springer, January 2008.

[13] R. Abbott, “Program design by informal English descriptions,” *Communications of the ACM*, vol. 26, no. 11, pp. 882–894, November 1983.

[14] J. Goyvaerts and S. Levithan, *Regular Expressions Cookbook*. O'Reilly Media, May 2009.

[15] G. Miller, “WordNet: A Lexical Database for English,” *Communications of the ACM*, vol. 38, no. 3, pp. 39–41, 1995.

[16] U. Frank, “Some Guidelines for the Conception of Domain-Specific Modelling Languages,” in *Proceedings of the 4th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA)*, September 2011, pp. 93–106.

[17] C. Hood, S. Wiedemann, S. Fichtinger, and U. Pautz, *Requirements Management: The Interface Between Requirements Development and All Other Systems Engineering Processes*. Springer, December 2007.

[18] S. Bradner, “RFC 2119: Key words for use in RFCs to Indicate Requirement Levels,” [online] Available at: <http://www.ietf.org/rfc/rfc2119.txt> [Accessed January 25, 2013].