# SpecQua: Towards a Framework for Requirements Specifications with Increased Quality

Alberto Rodrigues da Silva

Instituto Superior Técnico, Universidade de Lisboa & INESC-ID, Lisbon, Portugal
alberto.silva@tecnico.ulisboa.pt

**Abstract.** Requirements specifications describe multiple technical concerns of a system and are used throughout the project life-cycle to help sharing a system's common understanding among multiple stakeholders. The interest to support the definition and the management of system requirements specifications (SRSs) is evident by the diversity of many generic and RE-specific tools. However, little work has been done in what concerns the quality of SRSs. Indeed, most recommended practices are mainly focused on human-intensive tasks, mainly dependent on domain experts, and so, these practices tend to be time-consuming, error-prone and unproductive. This paper proposes and discusses an innovative approach to mitigate this status, and defends that with proper tool support – such as the SpecQua framework discussed in the paper –, we can increase the overall quality of SRSs as well as we can increase the productivity associated to traditional tasks of RE such as documentation and validation.

**Keywords:** Requirements Specification, Quality of Requirements Specification, Requirements Validation.

## 1. Introduction

Requirements Engineering (RE) intends to provide a shared vision and understanding of the system to be developed between business and technical stakeholders [1,2,3]. The adverse consequences of disregarding the importance of the early activities covered by RE are well-known [4,5]. System requirements specification, software requirements specification or just requirements specifications (SRS) is a document that describes multiple technical concerns of a software system [1,2,3]. An SRS is used throughout different stages of the project life-cycle to help sharing the system vision among the main stakeholders, as well as to facilitate communication and the overall project management and system development processes. For achieving an effective communication, everyone should be able to communicate by means of a common language, and natural language provides the foundations for such language. Natural language is flexible, universal, and humans are proficient at using it to

communicate with each other. Natural language has minimal adoption resistance as a requirements documentation technique [1,3]. However, although natural language is the most common and preferred form of requirements representation [6], it also exhibits some intrinsic characteristics that often present themselves as the root cause of many requirements quality problems, such as incorrectness, inconsistency, incompleteness and ambiguousness [1,3]. From these causes, in this paper we emphasize inconsistency and incompleteness because avoiding – or at least mitigating – them requires significant human effort due to the large amount of information to process when combined with inadequate tool support, namely to perform the typical requirements linguistic analysis. On the other hand, although ambiguity and incorrectness – by definition – cannot be fixed without human validation [7], we consider that the tasks required to minimize the effects of both inconsistency and incompleteness (and also ambiguity at some extent) can be automated if requirements are expressed in a suitable language, and if adequate tool support is provided.

In our recent research we consider the RSLingo approach [9,10,11] as a starting point for this challenge. RSLingo is an approach for the formal specification of software requirements that uses lightweight Natural Language Processing (NLP) techniques [8] to translate informal requirements – originally stated in unconstrained natural language by business stakeholders – into a formal representation provided by a language specifically designed for RE. Unlike other RE approaches, which use languages that typically pose some difficulties to business stakeholders (namely graphical modeling languages such as UML or SysML, whose target audience are engineers), RSLingo encourages business stakeholders to actively contribute to the RE process in a collaborative manner by directly authoring requirements in natural language. To achieve this goal, RSLingo provides (1) the RSL-PL language for defining linguistic patterns that frequently occur in requirements specifications written in natural language, (2) the RSL-RSL language that covers most RE concerns, in order to enable the formal specification of requirements, and (3) an information extraction mechanism that, based on the linguistic patterns defined in RSL-PL, translates the captured information into a formal requirements specification encoded in RSL-IL [10]. However, RSLingo does not provide yet any guarantees that the RSL-IL specifications have the required quality. Therefore, *the main contribute of this research is to propose and discuss that, with proper tool support and an intermediate representation language such as RSL-IL, we can increase the overall quality of SRSs as well as the productivity associated to documentation and validation tasks*. To the best of our knowledge, and apart from our own research on this issue [12], no further works have been proposed before in relation to this complex approach and the way we support automatic validation of SRSs as well as we have discussed the subject of modularity, dependencies and combinatorial effects at requirements level [13].

The structure of this paper is as follows. Section 2 introduces the background underlying this research. Section 3 overviews the SpecQua framework. Section 4 introduces some more technical details of the SpecQua with the purpose to show and discuss the practicability of the proposed approach. Section 5 discusses some experiments that are being implemented in the context of the SpecQua, in particular related to consistency, completeness and unambiguousness. Finally, Section 7 presents the conclusion and ideas for future work.

## 2. Background

This section briefly introduces the definition for SRS's quality attributes, overviews requirements specification languages, introduces some considerations on requirements validation, and briefly introduces the RSLingo approach.

### 2.1 SRS's Quality Attributes

Writing good requirements is a human-intensive and error prone task. Hooks summarize the most common problems in that respect [14]: making bad assumptions, writing implementation (How) instead of requirements (What), describing operations instead of writing requirements, using incorrect terms, using incorrect sentence structure or bad grammar, missing requirements, and over-specifying. To achieve quality SRSs must embody several characteristics. For example, the popular "IEEE Recommended Practice for Software Requirements Specifications" states that a good-quality SRS should be [7]: correct, unambiguous, complete, consistent, prioritized, verifiable, modifiable, and traceable. From all of them, we briefly discuss those that are most relevant for the scope of this paper.

**Complete**. A SRS is considered *complete* if it fulfills the following conditions: (1) Everything that the system is supposed to do is included in the SRS; this can lead us to a never ending cycle of requirements gathering; (2) Syntatic structures filled, e.g.: all pages numbered; all figures and tables numbered, named, and referenced; all terms defined; all units of measure provided; and all referenced material present; and (3) No sections or items marked with "To Be Determined" (TBD) or equivalent sentences. Completeness is probably the most difficult quality attribute to guarantee. In spite that some elements are easy to detect and correct (e.g., empty sections, TBD references), but one never knows when the actual requirements are enough to fully describe the system under consideration. To achieve completeness, reviews of the SRS by customer or users are essential. Prototypes also help raise awareness of new requirements and help us better understand poorly or abstractly defined requirements.

**Consistent**. A SRS is *consistent* if no requirements described in it conflict among themselves. Disagreements among requirements must be resolved before development can proceed. One may not know which (if any) is consistent until some research is done. When modifying the requirements, inconsistencies can slip in undetected if only the specific change is reviewed and not any related requirements.

**Unambiguous**. A SRS is *unambiguous* if every requirement stated there has only one possible interpretation. The SRS should be unambiguous both to those who create it and to those who use it. However, these groups of users often do not have the same background and therefore do not tend to describe software requirements the same way. Ambiguity is a very complex phenomenon because natural language is inherently ambiguous (a simple word can have multiple meanings) and most of the times this ambiguity is unintentionally introduced. The most recommended solution to minimize ambiguity is the use of formal or semi-formal specification languages rather than or in complement to natural languages. Also, the use of checklists and scenario-based reading are common recommendations [15].

### 2.2 Requirements Specification Languages

Traditionally, the requirements documentation activity has consisted in creating a natural language description of the application domain, as well as a prescription of what the system should do and constraints on its behavior [16]. However, this form of specification is both ambiguous and, in many cases, hard to verify because of the lack of a standard computer-processable representation [17].

Apparently, the usage of formal methods could overcome these problems. However, this would only address part of the problem, as we still need to take care while interpreting the natural language requirements to create a formal specification, given that in general engineers often misinterpret natural language specifications during the design phase. The same occurs with the attempt to directly create formal requirements specifications, especially when the real requirements are not discovered and validated at first by the business stakeholders [18]. Thus, the usage of such formal languages entails an additional misinterpretation level due to the typically complex syntax and mathematical background of formal method languages [17]. Given that formal methods are expensive to apply – because they require specialized training and are time-consuming [2] –, creating formal requirements specifications might have a negative impact.

In the attempt of getting the best from both worlds – the familiarity of natural language and the rigorousness of formal language –, one can document requirements with controlled natural languages, which are languages engineered to resemble natural language. However, these languages are only able to play the role of natural language to a certain extend: while they are easy to read, they are hard to write without specialized tools [19,20].

Finally, there are graphical approaches, such as UML and SysML for traditional RE modeling, and i*, KAOS and Tropos notation for Goal-Oriented RE [1]. However, these graphical languages are less expressive than natural language and cannot be regarded as a common language to communicate requirements, because business stakeholders still require training to understand them. Also, despite being "easier to understand" than formal method languages, these graphical modeling languages are regarded as less powerful in terms of analytical capabilities because they often lack tool support to enforce the implicit semantics of their modeling elements, or might even intentionally leave some unspecified parts of the language itself to ease its implementation by tool vendors, in which case they are considered as semi-formal. Some authors even argue that the simplicity of these languages comes precisely from this lack of semantic enforcement: it is easy to create models because "anything goes" [5].

Furthermore, the usage of graphical languages might cause another problem when the modeler includes too much detail in the diagram, cluttering it and thus affecting its readability. Therefore, despite the existence of such graphical approaches, textual natural language specifications are still regarded by many as the most suitable, fast, and preferred manner to initiate the requirements development process of the envisioned software system.

### 2.3 Tools and Interoperability Formats

Regarding the documentation and management of requirements there are two main approaches depending on the respective tool support. On one hand, the approach based on *generic tools (usually office applications and suites, word processors, or just text editors)* such as MS-Word, MS-Excel, OpenOffice or GoogleDocs. This is the most popular approach and many times the solution for simple projects: these tools are ubiquitous and everyone knows how to use them; they are flexible and good enough for most documentation needs. However, these tools do not provide specific features, particularly in what concerns the validation and management of requirements. On the other hand, *RE-specific tools* – such as Doors, RequisitePro, ProR, CaliberRM or Visure Requirements –, allow to define each requirement as a set of attributes, can establish link requirements with each other, and offer analysis and reporting features. Additionally, some tools allow for concurrent editing and change tracking, and to establish links between textual requirements and visual models represented in languages such as UML, SysML or BPMN. Depending on these approaches there are different formats to support requirements interoperability between tools. The common formats provided by generic tools are TXT (plain text) and RTF (rich text format) while RE-specific tools provide XML-based formats such as XMI (usual in UML CASE tools) and ReqIF (Requirements Interchange Format).

### 2.4 Requirements Validation

There is not a consensus in the literature about the use of the terms "verification" and "validation" in the context of RE. However, in this paper we adopt the term as suggested by Pohl and Robertsons, who define *requirements validation* as checking requirements with the purpose of detecting errors such as inconsistencies, ambiguities, and ignored standards. These authors recommend the use of the term "verification" to denote the formal (mathematical) proof of properties of a model, related to properties of concurrent systems, such as safety or absence of deadlocks [1,2]. Considering the premises regarding the current practices of the requirements documentation and validation activities – such as inspections, walkthroughs, checklists, or using scenarios and prototypes [1,2,3] –, we consider that the quality of a SRS 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 one requires an experienced requirements engineer with a vast skills set. However, to avoid large discrepancies in the results of the RE process, we advocate that the quality of requirements specifications and the productivity of the requirements documentation activity can be increased through the formalization of requirements. The computer-processable requirements specifications that are obtained through such a formalization process enable the automation of some manual validations thus relieving requirements engineers from the burden of manually handling a large amount of information to identify requirements quality problems. Additionally, the degree of formalization achieved can be employed to generate complementary artefacts to better support RE tasks, such as requirements validation.

### 2.5 The RSLingo Approach

RSLingo is an approach for the formal specification of software requirements that uses lightweight Natural Language Processing (NLP) techniques to (partially) translate informal requirements – originally stated by business stakeholders in unconstrained natural language – into a formal representation provided by a language specifically designed for RE [9].

The name RSLingo stems from the paronomasia on "RSL" and "Lingo". On one hand, "RSL" (Requirements Specification Language) emphasizes the purpose of formally specifying requirements. The language that serves this purpose is RSL-IL, in which "IL" stands for Intermediate Language [10]. On the other hand, "Lingo" expresses that its design has roots in natural language, which are encoded in linguistic patterns used during by the information extraction process [8,21] that automates the linguistic analysis of SRSs written in natural language. The language designed for encoding these RE-specific linguistic patterns is RSL-PL, in which "PL" stands for Pattern Language [11]. These linguistic patterns are used by lightweight NLP techniques and, when combined with general-purpose linguistic resources (e.g., VerbNet (http://verbs.colorado.edu/~mpalmer/ projects/verbnet.html) and WordNet (http://wordnet.princeton.edu)), enable the extraction of relevant information from the textual representations of requirements. Finally, the extracted information with these lightweight NLP techniques is formally specified in RSL-IL notation through predefined transformations from RSL-PL into RSL-IL. Upon a match of a requirement's textual representation with one of the RSL-PL linguistic patterns, a transformation should become active. This transformation takes into consideration the semantic roles of each word within the linguistic pattern, and drives the mapping between RSL-PL and RSL-IL.

RSL-IL provides several constructs that are logically arranged into viewpoints according to the specific RE concerns they address [10]. These viewpoints are organized according to two abstraction levels: business and system levels.

To properly understand and document the business context of the system, the business level of the RSL-IL supports the following business-related concerns, namely: (1) the concepts that belong to the business jargon; (2) the people and organizations that can influence or will be affected by the system; and (3) the objectives of business stakeholders regarding the value that the system will bring. Considering these concerns, business level requirements comprise respectively the following viewpoints: Terminology, Stakeholders, and Goals.

On the other hand, at the system level, the RSL-IL supports the specification of both static and dynamic concerns regarding the system, namely: (1) the logical decomposition of a complex system into several system elements, each with their own capabilities and characteristics, thus providing a suitable approach to organize and allocate requirements; (2) the requirements that express the desired features of the system, and also the constraints and quality attributes; (3) the data structures aligned with the business jargon, their relations, and a logical description of their attributes; and (4) the actors, functions, event-based state transitions, and use cases that further detail the aforementioned requirements. Considering these concerns, the System Level comprises the following viewpoints: Architectural; Requirements; Structural; and Behavioral, respectively.

## 3. SpecQua Framework – General Overview

Figure 1 illustrates the context diagram of the SpecQua framework and in particular shows the actors that might interact with it, namely: users (such as requirements engineers and business analysts), and tool superusers and administrators; and third-party tools that directly or indirectly interact with SpecQua. SpecQua provides several import and export features to guarantee the maximum interoperability with third-party tools, namely with commercial or open-source RE-specific tools and also general-purpose text-based authoring tools.

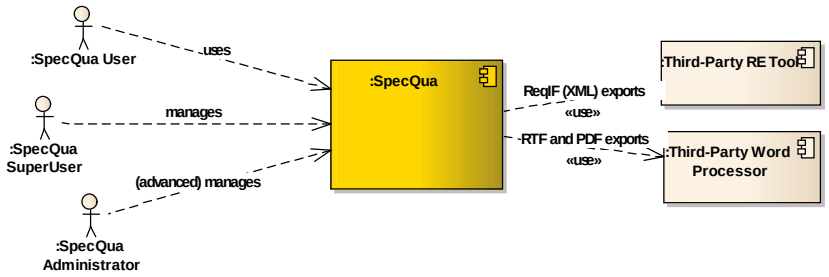


Figure 1 – SpecQua context diagram.

Figure 2 shows the common usage workflow supported by the SpecQua framework. This starts (1) by importing a set of requirements specifications produced in the context of third-party tools, namely by importing from one of the following formats: RSL-IL plain text, NL (natural language) plain text, or ReqIF. Then, users might (2) author the requirements directly in the SpecQua tool and (3) run a selected number of tests to automatically validate the quality of the involved requirements. Finally, if the requirements have the intended quality, (4) the requirements can be exported to third-party tools in formats such as ReqIF, RTF or PDF, and this process can iterate as many cycles as needed.

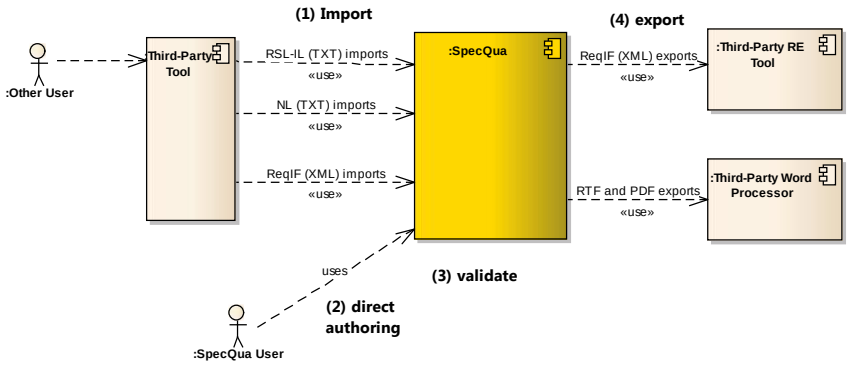


Figure 2 – SpecQua context diagram (with the common usage workflow).

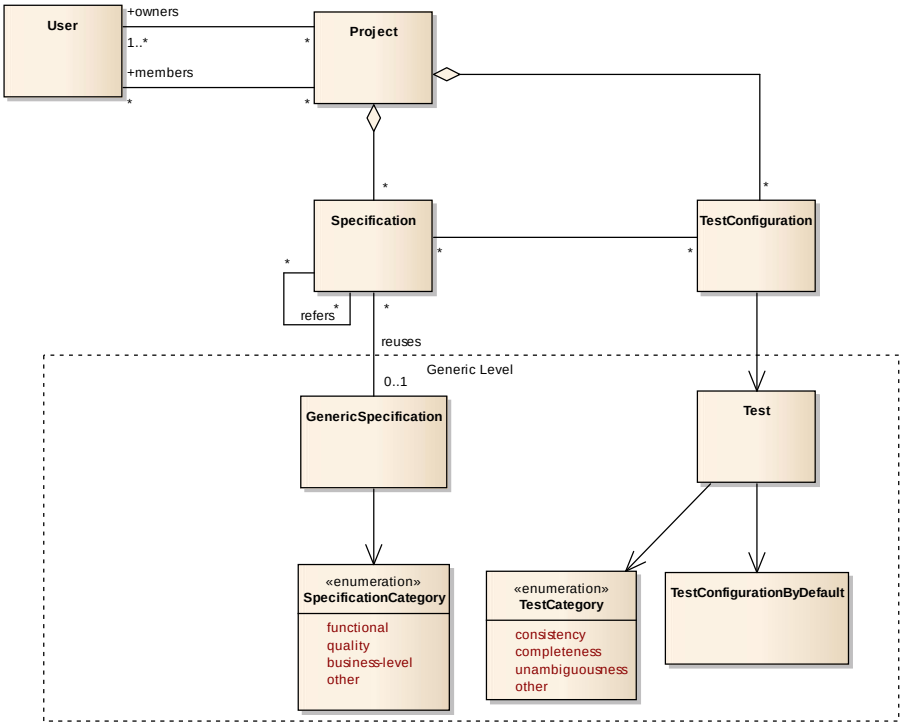


**Fig. 3.** SpecQua domain model.

The key concepts underlying the SpecQua tool are depicted in Figure 3 in a simplified way. There are two levels of abstraction in this domain model: project-level and generic-level. The concepts defined at . . . . . are managed by users with most responsibility (i.e., by the SpecQua SuperUsers), and involve the definition of GenericSpecifications and Tests. . . . . consists in a set of requirements (following the ReqIF terminology) that are defined in a project-independent way, so that they can be reused throughout different projects with minor changes. . . . . is a type of automatic validation that can be defined and run against one or more requirement specifications (see below for more information about Tests).

On the other hand, at . . . . . , there are common concepts to support the management of different projects, assign them to different users, with different permissions, and so on. At this level, any user can create a project, invite other users to participate in that project, and then start to import from third-party tools or author requirement . . . . . directly. Users can also establish relationships between specifications and reuse (by copy and customization) GenericSpecifications previously defined. Users may select and configure the relevant tests (via . . . . . ) in order to automatically validate their specifications. If these specifications have the required quality they can be exported to third-party tools or some kind of SRS report can be produced automatically.

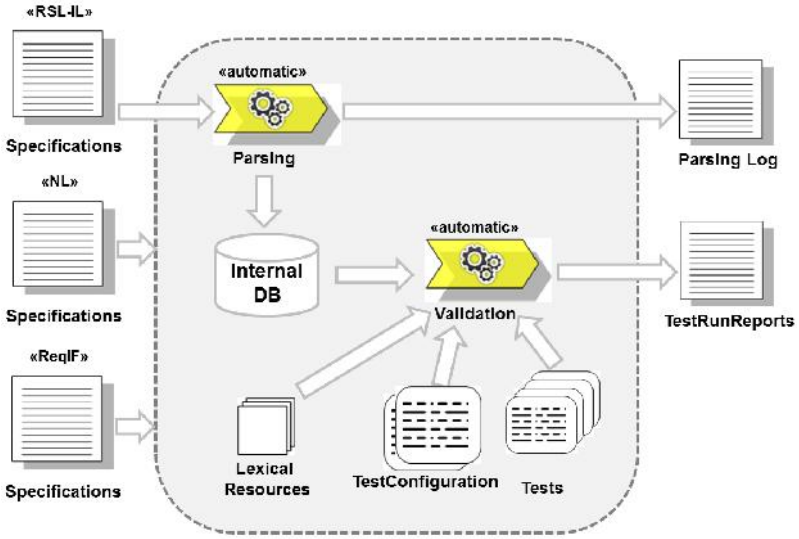


**Fig. 4.** Overview of the validation process.

Figure 4 suggests the general operation of the validation process with its main input, outputs, and supporting resources. The major input is the `Specifications` files that are the SRS defined in the RSL-IL concrete syntax (or the alternative formats, such as NL or structured ReqIF). The validation outputs are the `Parsing Log` file and the `TestRun Reports` file with the errors and warnings detected by the tool during its execution, respectively during the `Parsing` and the `Validation` processes. Additionally, there are some supporting resources used to better extend and support the tool at run-time, namely: `Tests`, `TestConfiguration`, and `Lexical Resources`.

`Tests` are directly implemented in a given programming language and have additional metadata such as name, description and quality category. (Figure 9 gives an example of such test directly implemented in C#.) `TestConfiguration` is a resource used to support the validation in a flexible way. For example, this resource allow requirements engineers to configure the level of completeness needed for their purpose, in a project basis. This means that different projects may have different needs regarding completeness for their specifications. Finally, `Lexical Resources` are public domain resources (such as WordNet or VerbNet) that support some tests, mostly those related with linguistic issues.

## 4. SpecQua Framework – More Details

The proposed high-level approach has to be implemented by a concrete software tool to offer a real interest and utility. Due to that we have implemented the SpecQua Framework with the purpose to show and discuss the practicability of this approach.

The SpecQua has the following objectives. First, provide SRS's quality reports: for a given RSL-IL specification, the system should be able to provide results for quality tests applied to that specification. Second, easily add and configure new quality tests: it should be easy to develop and add new quality tests in order to extend the tool; additionally it should be easy to configure those tests. Third, provide a Web collaborative workspace: the tool should be accessible via a Web browser and should provide the means for multiple users to work together while specifying, analyzing and validating RSL-IL specifications. (A preliminary prototype is available at http://specqua.apphb.com).

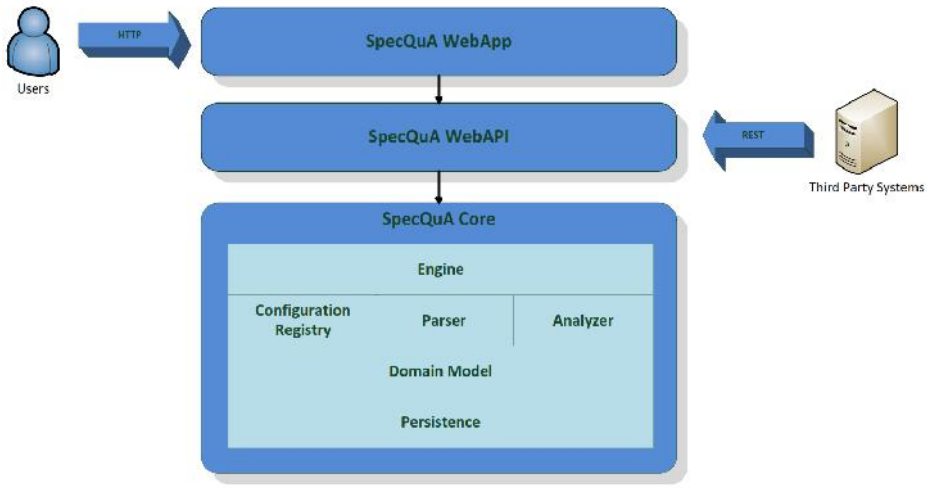


**Fig. 5.** SpecQua Architecture.

Figure 5 depicts the SpecQua software architecture in generic terms, with its main blocks: WebApp, WebAPI and Core.

**SpecQua WebApp**. The WebApp layer corresponds to how users interact with the tool. In a 3-tier layer architecture this corresponds to the presentation layer, although this is more than a simple frontend: the WebApp is a Web-based independent application with its own logic that connects to SpecQua Core via its API.

**SpecQua WebAPI**. This intermediate layer serves as the Application Programming Interface (API) that exposes all the relevant functions that Core provides to the outside. Besides serving as a proxy between WebApp and Core, the API can still be accessed from other clients that do not intend to use the main frontend but still want to take advantage of SpecQua analyses or data resources.

**SpecQua Core**. This layer is the kernel of the SpecQua tool and this is where all the action takes place. A key feature for the system SpecQua is the ease to add new

tests and this is implemented using two advanced programming techniques (i.e., dependency injection and reflection) which combined make it possible to associate new quality tests to the tool with minimal effort. At system startup, and based on referenced assemblies, all classes that implement a specific interface become available to the system as new quality tests, and are logically grouped into analyses. Currently, this grouping of testing analysis is performed using a configuration file as exemplified in Figure 6.

Each test is independent from each other and may have its own particular configuration. This configuration is defined in XML format and has no restriction on the schema level: it is up for those who develop the test (i.e., the SpecQua superuser), to define the schema and interpret it in the respective test. Each test may or may not have a default configuration.

```
<?xml version="1.0"?>
<analyses>
  <analysis id "completeness" name "Completeness" descrip-
tion="Completeness">
    <test id="comp-test-1"                type=" Spec-
QuA.Core.Tests.Completeness.ViewpointsElementsAndAttributesNotEmptyTes
t, SpecQuA.Core" />
    <test id="comp-test-2" type=" Spec-
QuA.Core.Tests.Completeness.StakeholdersAsGlossaryTermsTest, Spec-
QuA.Core" />
  </analysis>
  <analysis id="consistency" name="Consistency" descrip-
tion="Consistency">
    <test id="cons-test-1"                type=" Spec-
QuA.Core.Tests.Consistency.NoSynonmsAreReferencedOnSpecificationTest,
SpecQuA.Core" />
    <test id "cons-test-2"                type " Spec-
QuA.Core.Tests.Consistency.SynsetPropertyMustMatchExternalLexicalDatab
aseTest, SpecQuA.Core"/>
  </analysis>
  <analysis id="prioritization" name="Prioritization" descrip-
tion="Prioritization">
    <test id="pri-test-1"               type=" Spec-
QuA.Core.Tests.Prioritization.SpecificationHasPrioritiesCorrectlyDistr
ibutedTest, SpecQuA.Core"/>
  </analysis>
</analyses>
```

**Fig. 6.** Example of a SpecQuA's Test configuration file.

```
public specification returns [ProjectSpecification Specification]
  : spec = projectDef { $Specification = $spec.specification; };

projectDef returns [ProjectSpecification specification]
  : { $specification = new ProjectSpecification(); }
    LPAR PROJECT  (retAtt=attribute {CollectAttribute($specification, $retAtt.att.Key, … ;})*
   (LPAR gloss = glossary { $specification.Glossary = $gloss.glossary ;} RPAR)?
     (LPAR stk = stakeholders {$specification.Stakeholders = $stk.stakeholders;} RPAR)?
     (LPAR gls = goals {$specification.Goals = $gls.goals;} RPAR)?
     (LPAR system {} RPAR)?
   RPAR;
…
```

**Fig. 7.** An excerpt from the RSL-IL grammar for ANTLR.

Additionally, the SpecQua Core has the Parser component which is responsible to parse the input RSL-IL text and map it to the Domain Model (in the internal database). This parsing is done by ANTLR (http://www.antlr.org/) using a grammar specific for the RSL-IL language. With this grammar, ANTLR generates a parser and a lexer that can be used to validate the syntax and the semantic of a given input. If the input does not have errors, then the result is a representation of the domain model entities that can be tested. This grammar has some similarities with the BNF notation as expressed in Figure 7. (Other parsers to different import formats might be developed in the future.)


**Fig. 8.** Excerpt of a Report Test for the AIDSPortal example.

## 5. Discussion

Despite having a simple user-interface, a lot is done in the background when carrying out the analysis of a specification. In this case, when the user wants to validate the specification, it is parsed with the specific ANTLR grammar. If any syntactic or semantic error is detected in the specification, the user is alerted and the process stops. However, if the specification is successfully parsed, the tests can be selected and configured and are run against the specification. Finally, a report is shown to the user such as the example shown in Figure 8. In the following subsections we introduce and discuss some tests that are being implemented in the context of the SpecQua framework.

### 5.1 Consistency Validation

The consistency validation enforces that the information model underlying the RSL-IL specification is well-formed, or consistent in accordance with the RSL-IL metamodel, which involves, for example, the following concrete validations.

**Consistent attribute values**. SpecQua verifies whether the value assigned to a given attribute is valid based on the semantics of its RSL-IL construct. For instance, SpecQua can systematically enforce that every *id* attribute follows the predefined prefix of each RSL-IL construct. Also, SpecQua can also provide a short mnemonic for the *id* attribute based on the word attribute of the related Term, which is more meaningful than just a numeric sequence.

The combination of the specific prefix with this mnemonic allows one to better understand when two RSL-IL constructs of different types refer to the same underlying concept (e.g., they describe different concerns about the same business notion).

**Consistent numeric sequences**. There are several attributes in RSL-IL constructs that follow a certain numeric sequence. For instance, SpecQua checks the *order* attribute of each Sentence within a given Requirement. Also, SpecQua verifies the values assigned to the *label* attributes of a given UseCase's Steps. In all these cases, SpecQua must ensure that each construct was assigned a unique numeric value of that sequence, and that all the assigned numbers follow a monotonic increasing sequence without gaps.

**Referential integrity**. SpecQua must check and enforce that those relationships between different RSL-IL constructs are properly defined in terms of the values (i.e., references) assigned to the attributes that support the establishment of such relationships. The most obvious case is given by the strong dependency of most RSL-IL constructs on a Term that unambiguously defines the semantics of the underlying concept. Thus, SpecQua must check whether all RSL-IL constructs that depend on the Terminology viewpoint effectively provide a valid reference for a previously defined Term through its *id*. That is the example of the test shown in Figure 9 that checks if all Stakeholders are referenced as Terms defined in the glossary.

Another important aspect of this validation is also to support for the resolution of Term references based on their *acronym* or *word* values, instead of only relying on the value of its *id* attribute.

Although we are illustrating this sort of validations mostly based on the Terminology viewpoint, there are similar cases in other RSL-IL viewpoints. For instance, this problem is similar to the validation performed regarding the *source* attribute of a given Goal, which should be resolved to a well-defined Stakeholder.


**Fig. 9.** An excerpt of a concrete Test.

### 5.2 Completeness Validation

The completeness validation is based on the test's configuration resource that enables the definition of the level of completeness required for each RSL-IL specification. This level of completeness varies on a project basis or even, for the same project, along the timeline according the needs of the project team. We consider three levels of completeness:

**Completeness at model level**. At the macro level, one can define which viewpoints are required for considering a concrete RSL-IL specification to be complete. For example, during the initial stage of a project lifecycle, one may only require the Terminology, Stakeholders, and Goals viewpoints to consider the specification as being complete. On the other hand, if the project is running in a more advanced stage (for instance, after a couple of iterations), the remaining System Level viewpoints should be also considered in order to provide a complete requirements specification.

**Completeness at viewpoint level**. For each viewpoint one can define which constructs are mandatory or optional. For example, for the Behavioral viewpoint one might only consider as being relevant the existence of the Function construct (and not of Event) in order to consider that viewpoint as being complete.

**Completeness at construct level**. For each construct (e.g., Term, Stakeholder, Goal, Entity, Use Case) one can define which attributes are mandatory or optional. For example, for the Goal construct one can define the criticality as a mandatory attribute and the source (a reference for the Stakeholder responsible for that goal) as an optional attribute. Still at construct level, we can enforce that the names of some of these constructs (e.g., the names of actors and entities) should be defined as a unique term in the Terminology viewpoint.

### 5.3 Unambiguousness Validation

While in a formal specification (such as in RSL-IL) inconsistencies and incompleteness can be automatically detected, ambiguities deal directly with the meaning of those specifications, thus they are hard to be detected by automatic processes. Consequently, ambiguity tends to be detected mostly by human intervention, for example through analysis and inspection of the specification and through the use of prototypes. However, regarding this semantic level, still some automatic validation can be applied to reduce ambiguity, such as those discussed below.

**Semantic Analysis**. First, based on general-purpose linguistic resources that encode world knowledge, SpecQua can further verify the semantic validity of relations established between RSL-IL constructs, especially those strongly related with the natural language representation of concepts. For instance, an advanced validation feature consists in using WordNet to check whether the value of the word attribute of synonym Terms are indeed synonyms of the word attribute's value of the primary Term to which they are associated. Second, the information encoded within WordNet can be used to cross-check whether the Term associated with a given Stakeholder (through its role attribute) is aligned with the classification provided by the StakeholderType enumeration based on the *lexname* attribute of the WordNet synset referred by the *synset* attribute's value of that Term. Third, and still regarding the relations between different Stakeholders, SpecQua must verify the semantics of the hierarchical composition of these RSL-IL constructs. For instance, it does not make sense to specify that a Stakeholder whose type is "group.organizational" is MemberOf of another Stakeholder whose type is "individual.person". This means that the hierarchical Stakeholders composition must follow the implicitly semantics entailed in the values of the StakeholderType enumeration, which are ordered from broader groups to more specific entities. Fourth, SpecQua can determine whether the relation between a given RSL-IL construct and a Term is semantically valid based on the *pos* attribute of that Term and the semantics of the other RSL-IL construct. For instance, it does not make sense to associate an Entity with a Term whose part-of-speech (provided by either its *pos* or *synset* attributes) classifies the Term as a *verb*, instead of a *noun*. Fifth, another example consists in checking whether nouns associated with the agent thematic relation (e.g., the subject of natural language

sentences in the active voice) are defined as Actors and, if so, whether they can be traced back to the respective Stakeholders via a shared Term.

**Terminology Normalization**. The RSL-IL glossary (i.e., its Terminology viewpoint) formally defined the terms associated with the main concepts used throughout the requirements specification. There are different types of relations that can be established between terms, i.e. relations of type *synonym*, *antonym*, and *hyponym*. One motivation for using these relations is to reduce the number of redundant Terms employed within the RSL-IL specification, by providing a unique Term for each concept. So, it is important to avoid the definition of two or more synonym Terms by clearly stating which one of them should be classified as the *primary* Term, and the other(s) as *secondary* Term(s). Based on this information, SpecQua can perform a systematic normalization of Terms through a common find and replace process and, consequently, reduce the requirements specification's ambiguity.

## 6. Conclusion

RE comprises several tasks including requirements elicitation, analysis and negotiation, documentation and validation. We recognize that natural language is the most common and popular form to document SRSs. However, natural language exhibits several limitations, in particular those related with requirements specification quality such as incorrectness, inconsistency, incompleteness and ambiguousness.

This research extends the RSLingo approach by considering that requirements are represented in RSL-IL, automatically extracted from natural language specifications or authored directly by their users. This paper proposes a generic approach to automatically validate these specifications and describes the SpecQua framework that shows the practicability and utility of this proposal. The flexibility of SpecQua and the initial cases studies developed so far allows us to preliminary conclude that this approach helps to mitigate some of the mentioned limitations, in particular in what respect inconsistency, incompleteness and ambiguousness.

For future work we plan to implement SpecQua in a more extensive way, in particular with features related with the support of a collaborative environment, allowing end-users to author and validate directly their requirements [22], eventually with different representations beyond natural language, RSL-IL and ReqIF. Several import and export features would also be relevant in order to promote tools interoperability as discussed in the paper. Finally, we still intend to explore the integration of requirements specifications with testing [23] and model-driven engineering approaches [24,25,26] to still increase the quality and productivity of Software Engineering in general terms.

**Acknowledgements**

This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under the project PEst-OE/EEI/LA0021/2013 and DataStorm Research Line of Excellency funding (EXCL/EEI-ESS/0257/2012). Particular thanks to my students David Ferreira and Joao Marques for their strong participation and involvement in this research.

## References

1. Pohl, K., 2010. Requirements Engineering: Fundamentals, Principles, and Techniques, 1st edition, Springer.
2. Sommerville, I., Sawyer, P., 1997. Requirements Engineering: A Good Practice Guide. Wiley.
3. Robertson, S., Robertson, J. 2006. Mastering the Requirements Process, 2nd edition. Addison-Wesley.
4. Emam, K., Koru, A., 2008. A Replicated Survey of IT Software Project Failures. IEEE Software 25(5) (September 2008) 84–90.
5. Davis, A. M., 2005. Just Enough Requirements Management: Where Software Development Meets Marketing. Dorset House Publishing, 1st edition.
6. Kovitz, B., 1998. Practical Software Requirements: Manual of Content and Style. Manning.
7. IEEE Computer Society, 1998. IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830-1998.
8. Bird, S., Klein, E., Loper, E., 2009: Natural Language Processing with Python. O'Reilly Media, 1st edition.
9. Ferreira, D., Silva, A. R., 2012. RSLingo: An Information Extraction Approach toward Formal Requirements Specifications. In: Proc. of the 2nd Int. Workshop on Model-Driven Requirements Engineering (MoDRE 2012), IEEE CS.
10. Ferreira, D., Silva, A. R., 2013. RSL-IL: An Interlingua for Formally Documenting Requirements. In: Proc. of the of Third IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE 2013), IEEE CS.
11. Ferreira, D., Silva, A. R., 2013. RSL-PL: A Linguistic Pattern Language for Documenting Software Requirements. In: Proc. of the of Third International Workshop on Requirements Patterns (RePa 2013), IEEE CS.
12. Silva, A. R., 2014. Quality of Requirements Specifications: A Framework for Automatic Validation of Requirements, in Proceedings of ICEIS'2014 Conference, 2014, SCITEPRESS.
13. Silva, A. R., et al., 2014. Towards a System Requirements Specification Template that Minimizes Combinatorial Effects, Proceedings of QUATIC'2014 Conference, IEEE CS.
14. Hooks I., 1993. Writing Good Requirements, Proceedings of the Third International Symposium of the INCOSE, Volume 2.
15. Kamsties E., Berry D.M. and Paech B., 2001. Detecting Ambiguities in Requirements Documents Using Inspections, Proceedings of the First Workshop on Inspection in Software Engineering.
16. van Lamsweerde, A., 2009. From Worlds to Machines. In A Tribute to Michael Jackson. Lulu Press.
17. Foster, H., Krolnik, A., and Lacey, D., 2004. Assertion-based Design. Springer.
18. Young, R., 2003. The Requirements Engineering Handbook. Artech Print on Demand.

19. Fuchs, N. E., Kaljurand, K., Kuhn, T., 2008. Attempto Controlled English for Knowledge Representation. In Reasoning Web, Fourth International Summer School 2008, Lecture Notes in Computer Science, 5224, Springer.
20. Kuhn, T., 2010. Controlled English for Knowledge Representation. Ph.D. thesis, Faculty of Economics, Business Administration and Information Technology of the University of Zurich.
21. Cunningham, H., 2006. Information Extraction, Automatic. In Encyclopedia of Language & Linguistics, volume 5. Elsevier, 2nd edition.
22. Ferreira, D., Silva, A. R., 2008. Wiki supported collaborative requirements engineering, Proceedings of the 4th International Symposium on Wikis. ACM.
23. Moreira, R., Paiva, A. C. R., Memon, A., 2013. A pattern-based approach for GUI modeling and testing. In IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), IEEE CS.
24. Silva, A. R., et al., 2007. Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools. IET Software Journal,1(6), IET.
25. Savic, D., et al., 2012. Use Case Specification at Different Levels of Abstraction. In Proceedings of QUATIC'2012 Conference, 2012, IEEE CS.
26. Ribeiro, A., Silva, A. R., 2014. XIS-Mobile: A DSL for Mobile Applications. Proceedings of SAC 2014 Conference, ACM.