Measure Software Requirement Specifications by Ontology Reasoning

Katja Siegemund\textsuperscript{1}, Yuting Zhao\textsuperscript{2}, Jeff Z. Pan\textsuperscript{2}, and Uwe Aßmann\textsuperscript{1}

\textsuperscript{1} Technische Universität Dresden, Germany
\textsuperscript{2} University of Aberdeen, UK

Abstract. Requirements Engineering (RE) is essential to a software project. As the result of RE process, software Requirement Specifications (SRS) should be consistent, correct and complete. However, the acquisition, specification and evolution of requirements from different stakeholders or sources usually leads to incomplete, ambiguous, and faulty requirements. This may become an incalculable risk for the whole project and a disaster for the final software product. In this paper we present a method to improve the quality of a SRS semi-automatically. Facilitated by ontology reasoning techniques, we describe how to detect and repair faulty information in the SRS. Furthermore, we also provide various metrics to measure the quality of the SRS at any time during the RE process. Finally, we generalize our approach to be applicable for any information captured in an ontology.

1 Introduction

Software Requirements Engineering (RE) refers to the process of eliciting, evaluating, specifying, consolidating, and changing the objectives, functionalities, qualities, and constraints to be achieved by a software-intensive system \cite{1}. The result of a RE process is a software requirements specification (SRS) that describes all the externally observable behaviours and characteristics expected of a software system. Such a requirement specification is the foundation for software design decisions. Thus, it must be consistent, unambiguous, verifiable, and complete \cite{2,3}. The importance of RE was already identified in several studies (e.g. \cite{4}). Missing or incomplete requirements propagate through the entire system development and lead to incomplete or incorrect designs, architectures, implementations and tests. A poor SRS is often the main reason for project failures or increasing costs.

Inconsistencies result from the acquisition, specification, and evolution of goals and requirements from multiple stakeholders and sources \cite{5}. It is frequently the case that changes of requirements also have a particularly significant impact on the consistency of specifications. In order to regain consistency, requirements are removed from the specification which often leads to incompleteness. Zowghi et. al. (\cite{6}) describes this vicious circle as a causal relationship between consistency, completeness and correctness.

From a formal point of view, correctness is usually meant to be the combination of consistency and completeness. Therefore, the ability to detect and repair
inconsistent and incomplete requirements is crucial to the successful development of high-quality requirement specifications [7].

This paper is based on the Meta Model for Ontology-driven Requirements Engineering described in [7]. We extended the software prototype ONTOREQ (Ontology-Driven Requirements Engineering) to improve the quality of requirement specifications and to measure a set of quality attributes and to calculate the entire quality of the SRS. In this paper we illustrate how to improve the quality of SRS by using NBox reasoning with closed world assumption. Therefore, we use a continuous example for and provide several code snippets that demonstrate the interaction of reasoning technology, OWL DL and SPARQL. Additionally, we provide general domain independent guidelines to adopt our approach.

The paper is structured as follows: section 2 describes related work and the meta model our approach is based on. In section 3 we conceptually describe how to improve the quality of software requirement specifications. Domain independent guidelines are described in section 4. The technical realisation and a continuous example are provided in section 5. Finally, we summarize our approach and discuss further work in section 6.

2 Related Work

Several approaches exists that propose the application of ontological techniques to requirements engineering. However, most of them concentrate only on specific requirement artefacts, e.g. goals, requirements or use-cases and do not support reasoning over relations between all concepts.

Sancho et. al. ([8]) describe a method for ontology-aided performance engineering. They propose the structure of an ontological database to relate Performance Engineering issues among themselves and with other non-functional requirements. The NFR/i* Framework ([9]) focuses on qualitative reasoning for goal contribution and evaluation of alternative goal refinements. The KAOS Framework ([10]) concentrates mainly on goal satisfaction. Lamsweerde describes in [1] an interesting method for reasoning about alternative requirements and inspired some of the requirement relationships and ontology properties in our approach.

Other approaches aim to address a broader field of RE. The OntoRem approach ([11]) describes the development of a meta model for general RE concepts. Lin et. al. ([12]) developed a requirements ontology that includes parts, features, requirements, and constraints. They use querying for generating answers to common RE questions. Lee et. al. present an Ontology-based active Requirements Engineering Framework (Onto-ActRE) based on Goal-oriented RE [13]. They support relationships of requirements with other concepts, such as goals and risks, but do not consider the variety of relations between requirements.

3 Improving the Quality of Requirement Specifications

Generally, a SRS of high-quality is one that contributes to a successful and cost-effective creation of software that solves real user needs [14]. Davis in
[14] proposes 24 criteria to ensure the quality of a requirement specification. Some of the most important are: completeness, correctness, verifiability, internal consistency, conciseness and traceability. As already indicated in section 1 we base our approach on the ODRE (Ontology-Driven RE) meta model proposed in [7]. The main concepts of RE (RE artefacts) are modelled as OWLClasses (e.g. Goal, Requirement, UseCase, Author) in the Requirements Ontology. Of significant importance are the various object and data properties that interrelate requirements among each other and with other RE artefacts. These relations enable meaningful reasoning and queries. The concrete RE knowledge (ABox) is kept in the Requirements Ontology by instantiating the TBox. However, the requirements engineer will not have to work directly with the ontology since this is kept in background. A user-interface enables the modification of RE knowledge captured in the Requirements Ontology.

In the following, we describe our approach for ensuring, measuring and improving the quality of a software requirement specification.

3.1 Quality Check

Similar to the completeness and consistency rules proposed in [7] we define nine quality rules to be met. These rules are based on some of the quality attributes described by Davis in [14] and the knowledge specified in the Requirements Ontology. We define a quality rule as follows:

**Definition: Quality Rule**
A quality rule states essential requirement information to ensure the quality of a SRS and provides specific suggestions for every rule violation detected. It consists of a rule description, a fault message and a solution suggestion.

Obviously, an incomplete or inconsistent requirement specification cannot be of high-quality. Therefore, we demand the relevant completeness and consistency checks proposed in [7] before executing the quality check.

We specified nine quality rules to test for the above stated quality attributes. Here, we exemplarily present a selection of these quality rules which are of importance for describing and understanding the application of ontology reasoning techniques and for.

1. *Requirements must be complete.*
   This rule refers to the completeness rules. If incomplete but mandatory information is identified, it is suggested to execute the completeness check before proceeding with improving quality.

2. *The requirement configuration must be consistent.*
   This rule refers to the consistency rules. If inconsistent information is identified, it is suggested to execute the consistency check before proceeding with improving quality.
3. Each requirement must state its mandate (optional or mandatory).
   This allows further decisions on the inclusion or exclusion of requirements.

4. Requirements with a positive contribution to a challenge or goal should be included in the requirement configuration.
   This ensures goal satisfaction. Finally, it is the decision of the requirements engineer whether to include these additional requirements or not.

5. There must be no requirement that is a negative contribution to a goal to be achieved.
   Negative contributions to goals limit the quality of the SRS and therefore should not be included. The solution suggestion will display appropriate alternative requirements.

6. There must be no optional requirement with a high risk or high cost.
   Requirements with high risk or high cost may have a bad influence of the project and thus may not need to be implemented if optional anyway. The solution suggestion will also display alternative requirements instead.

3.2 Measuring SRS Quality

The above described quality rules involve the requirements engineer for improving the quality of the software requirement specification. Although these rules help to exactly identify where improvements are necessary, the requirements engineer will not know how meaningful the modifications were. Therefore, we provide a set of metrics that enable to automatically calculate the quality of the SRS.

In addition to the quality attributes of Davis, we define a new attribute, “uncritical”. Some of the definitions and most of the computations given in [14] have been modified by us to keep them up-to-date with today’s requirement engineering knowledge and to make them applicable for ontology reasoning techniques.

In the rest, we give a list of these quality criteria and their definitions. We also provide the magnanimity $Q_i$ and weight $W_i$ for each of them in ONTOREQ.

[Internal Completeness of SRS] The requirement specification is internally complete, if all necessary requirement artefacts, metadata for requirements and requirement relationships have been specified. Internal completeness is reached if the requirement configuration complies to all completeness rules.

Obviously, a requirement specification will never be totally complete, due to new arising and changing requirements. Moreover, it is not possible to state whether a requirement specification is complete, since we never know which requirements are forgotten. [14] describes this as “we don’t know we know these [requirements]”. Nevertheless, we can say whether all requirements metadata and requirement relationships have been completely specified. We use the following function:

$$Q_i = \frac{r_c}{r_n}$$
where \( r_c \) denotes complete requirements and \( r_n \) all requirements in the SRS. Values range from 0 (100% incomplete) to 1 (100% complete). Since internal completeness is an attribute of main importance, we use a weight \( W_1 = 1 \) in accordance to [14].

**[Correctness of SRS]** The percentage of validated requirements.

Like completeness, it is not trivial to measure the correctness of a requirement specification. Davis proposes to measure instead the correctness of individual requirements to make assumptions regarding the correctness of the entire requirement specification. An individual requirement is correct, if it “represents something that is required of the system to be built” [14]. Since we do not know which requirements are incorrect (otherwise we could easily correct them and would always reach 100% correctness), we will rather measure the percentage of requirements that have been validated. Note that we assign a property “invalid” to each newly specified requirement that has not been validated so far.

Thus, we assume that the requirements have previously been reviewed regarding their correctness during the validation task. Certainly, this method is not perfectly satisfying, but it is yet a practical approximation to make a statement about correctness. We adopt the following measure from Davis:

\[
Q_2 = \frac{r_v}{r_n}
\]

where \( r_v \) is the number of valid (and thus correct) requirements. Since correctness is so critical to project success, we use a weight of \( W_2 = 1 \), according to [14].

**[Verifiable SRS]** A requirement specification is verifiable if a test-case and/or metric is specified for each requirement.

A SRS can only be verified if it provides means to verify the individual requirements specified within. In ONTOREQ we require test-cases and/or metrics for each requirement. We use the following measure to calculate the percentage of verifiable requirements:

\[
Q_3 = \frac{r_v}{r_n}
\]

where \( r_v \) is the number of requirements with a test-case and/or metric. We adopt the weight of \( W_3 = 0.7 \) as recommended by Davis in [14].

**[Internal Consistency of Requirement Configuration]** A *requirement configuration* is internally consistent if it is free of conflicts and excluding requirements. All mandatory requirements and applicable coexistent requirements must be included. The requirement configuration must contain the most refined requirements of each particular requirement refinement.

Instead of only measuring the absence of conflicts as proposed by Davis, we define internal consistency much stronger. Therefore, we concentrate on the
requirement configuration rather than the whole specification. The requirement configuration is a selection of requirements from the requirement specification to be finally implemented. Thus, it can be checked against more consistency rules (e.g. whether mandatory requirements have been included) than the whole requirement specification. Furthermore, requirements in the SRS that have been identified as conflicting are actually no problem as long as they are not included in the final requirement configuration. So we propose the following function for measuring the internal consistency of the requirement configuration:

\[ Q_4 = \frac{r_c}{r_{rc}} \]

where \( r_{rc} \) is the number of consistent requirements in the requirement configuration. In accordance to Davis we weight this function with \( W_4 = 1 \) due to its importance for the overall quality of the requirement specification.

[Traceability of SRS] The requirement specification is traceable if it facilitates the traceability of requirement artefacts, especially of each requirement [14]. A requirement (artefact) is traceable if its life from analysis to code can be followed up. This requires at least the specification of its source, an author, the goal it contributes to, a use-case where it is described and a test-case and/or metric for verification.

In order to measure the traceability of a requirement specification, we need to be able to trace each single requirement. Therefore, we extended the definition proposed by Davis. Thus, we need to check for the completeness of the metadata required in the definition above. We use the following formula:

\[ Q_5 = \frac{r_t}{r_{rc}} \]

where \( r_t \) denotes the requirements that are traceable and \( r_{rc} \) is the number of all requirements in the requirement configuration. We will use the weight \( W_5 = 0.6 \) since traceability is not of main importance for the quality of the SRS.

[Uncritical Requirement Configuration] A SRS is all the more uncritical if individual requirements bear only few high risks and high costs.

We propose to measure the criticality of a SRS as an extension of the quality attributes describes by Davis ([14]). Certainly, a requirement specification will always contain requirements that are cost extensive or bear a high risk. We do not want to misapply this fact. However, the quality of a requirement specification is increasing if costs and risks are not ignored and a tendency can be reasoned. Thus, we measure the percentage of critical requirements. Just like for internal consistency, risky and cost-intensive requirements only become a problem if they are included in the requirement configuration, which is checked by the quality rules. We will use the following function to give the requirements engineer a feeling for the whole requirement specification.
Measure SRS by Ontology Reasoning 7

\[ Q_6 = \frac{r_r}{r_{rc}} \]

where \( r_r \) is the number of risky requirements and \( r_{rc} \) the number of all requirements in the requirement configuration. We recommend to weight this function with \( W_6 = 0.5 \).

**The overall SRS quality**

These six quality functions can be either used as single metrics for the above quality attributes or as well for calculating the overall quality of the SRS. Therefore, we propose the following function:

\[ Q = \frac{\sum_{i=1}^{6} W_i Q_i}{\sum_{i=1}^{6} W_i} \]

4 General Guidelines for Improving Quality of Knowledge in Ontologies

Our approach in section 3 to identify incomplete and inconsistent information and to calculate the quality of information provided is specific to the Requirements Ontology (TBox). However, the quality of data is important for every domain. Furthermore, quality generally requires completeness, consistency and correctness of the specified information. Thus, we provide some domain independent guidelines to apply our approach for any other ontology and the knowledge specified within. The code snippets provided in section 3 demonstrate implementation issues.

We propose the following guidelines to build an ontology TBox that can be reasoned for our purposes (completeness, consistency and quality).

1. Build a domain-specific ontology Meta Model (TBox) for storing the concrete knowledge as instantiation of it (ABox).

   The most important part of the Knowledge Repository are object and data properties. Only by using these properties we can interrelate instances of the classes and finally use this information for reasoning for various suggestions on how to eliminate problems. Thus, we carefully have to specify and use these properties.

2. Use ontology classes and properties to allow meaningful relations between important ontology artefacts.

   Although most of these properties will also be domain dependent, they must allow to check for completeness, consistency and quality. Therefore, we propose the following guideline:
3. The Ontology Meta Model must enable to reason about alternative, optional, mandatory, conflicting, excluding and coexisting individuals. If appropriate refinements shall be identifiable. For this purpose object and data properties must be utilized.

Further properties to specify cost, priority and so on may also be applicable. All of these interrelations will be used to generate solutions for revising information of low quality. The benefit and applicability of these suggestions only remains on the information provided in the ontology.

In order to identify information that is incomplete, inconsistent or a quality flaw in the ABox, we need to interpret these terms for a given domain and set up a set of rules the data must comply to. Thus, we have to define domain specific quality rules.

4. Identify domain specific quality attributes for the information specified in the domain specific Meta Model.
5. Define Quality Rules (in natural language or pseudo-code) to test for these quality attributes.
6. Implement these rules using any of the strategies described in section 3.

The calculation of the quality of information provided by an ontology is also domain specific. Just as specifying the quality rules, one has to decide what exactly influences the quality of information contained in the ontology besides the general applicable completeness and consistency.

7. Identify and define how to measure the quality (quality metrics) as described in section 3. Adapt the ontology if necessary to enable these calculations.
8. Define weights for the quality metrics.
9. Implement the calculation of quality as explained in section 5.

5 Technical Solution

In Section 3 we explained the concepts for checking and improving quality as well as how to measure the quality of the SRS. In this section we present several possible solutions and discuss associated problems. We provide a short example and code snippets to illustrates different solutions to this problems. Finally, we explain our technical solution for ONTOREQ.

5.1 Open World vs. Closed World

As already described in sections 1 and 3, the Requirements Ontology as meta model (TBox) is the fundament for our approach. It provides the necessary structure to specify and organize requirement artefacts and to interrelate them. When instantiated (ABox) it contains all requirement knowledge. Information that is not explicitly existing in an ontology is only assumed as “unknown” and thus, true, rather than “not existing” and false. This is due to the open world
**assumption (OWA)** where a deductive reasoner will not infer that the statement is false. Hence, in order to identify missing information we have to use **closed world assumption (CWA)** which holds that any statement that is not known to be true is false.

But RE is a continuing process. During the whole software development process it is possible that new requirements are identified or existing ones have to be modified in any way. So CWA never seems to be a solution.

Summarized, we need OWA for the process of specifying requirements and CWA for accomplishing completeness, consistency and quality checks of the Requirements Ontology. Thus, we need to switch between OWA and CWA. This way, we can for example specify a number of requirements (OWA), then check for their completeness (CWA), identify that we forgot to specify something (CWA), add some data (OWA) and check again for completeness (CWA). Since we are interested in identifying information that is not existent, we have to use NBox reasoning. An NBox (Negation As Failure Box) is a set of classes and object properties, whose extensions are closed. If an individual is not inferred to be an instance of a closed class, then it is regarded as an instance of the negation of that closed class [15].

In the following we describe two techniques to support reasoning for incomplete information and explain advantages and disadvantages. Therefore, we use the following brief example knowledge (ABox) of the Requirements Ontology and an extract of two quality rules. We use FR as abbreviation for functional requirement.

<table>
<thead>
<tr>
<th>Requirements knowledge in the Requirements Ontology:</th>
<th>Expected Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1, FR2, FR3, Goal1, Goal2, Author1</td>
<td>Requirements with no author: FR1, FR3</td>
</tr>
<tr>
<td>FR1 isAlternativeTo FR2</td>
<td>FR1 has a negative contribution to Goal2</td>
</tr>
<tr>
<td>FR1 isNegativeContributionTo Goal2</td>
<td></td>
</tr>
<tr>
<td>FR2 isAuthoredBy Author1</td>
<td></td>
</tr>
</tbody>
</table>

**Quality Rules to be checked:**

1. Requirements must be complete.

2. There must be no requirement that is a negative contribution to a goal to be achieved.

**Expected Result:**

- Requirements with no author: FR1, FR3
- FR1 has a negative contribution to Goal2.

All of the solutions given here have the following prerequisites in common: For accessing and computing the Requirements Ontology (concepts, properties and individuals), we need a Java Application. Ontologies can be accessed and modified by using the owlapi. This is a trivial task and will not be explained here. Additionally, we use the following prefix for each of the SPARQL queries listed below:

```sparql
ro:<http://www.semanticweb.org/ontologies/2012/4/ro.owl#>
```

**Bypassing Closed World Reasoning with Java and OWLAPI2** We can bypass closed world reasoning by using owlapi2 for accessing the ontology and a reasoner of choice (e.g. Jena, Pellet) to reason about the Requirements Ontology.
and to identify missing information. One possible solution for bypassing closed world reasoning is based on the following strategy. We retrieve all requirements and for each requirement the relation of a specific OWLObjectProperty. The quantity of these relations for one individual and one kind of OWLObjectProperty can be accessed via the size() method provided by OWLObjectPropertyValues. It returns a set of individuals that are the values of this property. Thus, an empty set is the evidence for missing information. The following code snippet illustrates bypassing.

```
Listing 1.1: Bypassing Closed World Reasoning

public Set<String> identifyMissingAuthors(Set<OWLNamedIndividual> indSet) {
    Set<String> resultList = new TreeSet<String>();
    if (!indSet.isEmpty()) {
        for (OWLNamedIndividual i : indSet)
            if (reasoner.getObjectPropertyValues(i, isAuthoredBy).getFlattened().size() == 0)
                resultList.add(i.getIRI().getFragment());
    }
    return resultList;
}

System.out.println("The following requirements have no author specified: " +
    identifyMissingAuthors(requirementInd).toString());
```

The method `identifyMissingAuthors` returns the values of the OWLObject-Property `isAuthoredBy` for each requirement in a set of OWLNamedIndividual (`indSet`). In this example, the property values are the authors of a requirement, e.g. author1, author2. If the returned values are equal to 0 (no authors exist), then this requirement is added to the `resultList` that contains all requirements without an author specified.

Local Closed World Reasoning (LCWR) with Negation as Failure

Far more intuitive and convenient than bypassing closed world reasoning is the use of SPARQL queries with negation as failure and CWA. Thus, we can explicitly ask for information that is not existent in the Requirements Ontology. Therefore, we need the following prerequisites: a reasoner enabling LCWR (e.g. TrOWL [15]), owlapi and SPARQL 1.1. The strategy for identifying missing information with CWA and negation is as pretty intuitive. We close all concepts in the ontology we want to reason about and use SPARQL to build queries that extract all requirements without a specific information.

```
Listing 1.2: Closing Axioms in the Requirements Ontology

File newFile = new File("C:/.../new_ro.owl");
OWLOntology newOnto = manager.createOntology(IRI.create(newFile));
for (OWLLogicalAxiom axiom : localRO.getLogicalAxioms())
    manager.addAxiom(newOnto, axiom);

OWLAnnotationProperty property = factory.getOWLAnnotationProperty(IRI.create("http://TrOWL.eu/REL#NBox"));
OWLAnnotation annotation = factory.getOWLAnnotation(property, factory.getOWLLiteral("close", "en"));
OWLAnnotationAssertionAxiom axiom = factory.getOWLAnnotationAssertionAxiom(requirement.getIRI(), annotation);
```

The method `identifyMissingAuthors` returns the values of the OWLObject-Property `isAuthoredBy` for each requirement in a set of OWLNamedIndividual (`indSet`). In this example, the property values are the authors of a requirement, e.g. author1, author2. If the returned values are equal to 0 (no authors exist), then this requirement is added to the `resultList` that contains all requirements without an author specified.
Lines 1 to 4 illustrate how to create a new ontology and copy all axioms from the Requirements Ontology. To specify which class/property names to be closed in an OWL ontology, TrOWL ([16]) uses the specified annotation property in line 6 and 7. Since we want to retrieve all individuals of the class requirements with no author, we have to close this OWLClass (line 8). In line 10 we add this new axiom to newOnto (localRO is the OWLOntology Object of the original Requirements Ontology) and save the ontology in line 11.

Listing 1.3: SPARQL Query for Missing Information

```java
String queryFRHasAuthor = "SELECT ?r WHERE {?r a ro:Requirement . FILTER NOT EXISTS {?r ro:isAuthoredBy ?a}}";
```

This SPARQL query returns all requirements without the OWLObjectProperty isAuthoredBy.

Listing 1.4: Identification of Missing Information

```java
public Set<String> performQuery(String query, String var, String artefact) {
    Set<String> resultNames = new TreeSet<String>();
    Query q = QueryFactory.create(query);
    QueryExecution qe = QueryExecutionFactory.create(q, ontModel);
    ResultSet rs = qe.execSelect();
    while (rs.hasNext()) {
        QuerySolution solution = rs.next();
        String name = solution.getResource(var).getLocalName();
        resultList.add(name);
    }
    for (String item : resultList) {
        System.out.println(item + " has no" + artefact + ".");
    }
    return resultList;
}
```

This code snippet demonstrates the execution of the SPARQL query and the output of incomplete requirements. The method performQuery executes a query query and binds all results for the variable var. The variable artefact is used for the output of the appropriate requirement artefact.

Using SWRL Rules and LCWR A good part of the quality rules and measurements check for the existence (or not existence) of multiple patterns. Suppose, we want to define important requirements as mandatory requirements with high priority. Thus, an important requirement must be mandatory and have a high priority. We have (at least) two options to enable reasoning for these important requirements: (1) using SPARQL queries to query for all requirements with the OWLDataProperty isMandatory and the OWLClassType HighPriority or (2) using SWRL rules in the Requirements Ontology itself to define an equivalent class ImportantRequirements and infer the relevant individuals to this class with SWRL rules. We will illustrate the second approach:
Having specified these rules in the Requirements Ontology, we can simply use a SPARQL query to query for individuals of the OWLClass ImportantRequirement:

```
SELECT ?r WHERE {?r a ro: ImportantRequirement} ;
```

We suggest to define such classes in the ontology itself rather than reasoning for multiple patterns. The advantages are manifold: we only need one single (and short) SPARQL query, the information in the ontology can be accessed either using queries or directly via owlapi and a reasoner that supports rules (e.g. Pellett) and, finally, the knowledge captured in the rules actually belongs to the knowledge repository for RE and thus, should not be hidden or built around in some programming code. However, we need to be aware of the fact that it is not possible to use negation or enumeration in swrl rules since it shares OWL’s open world assumption.

### 5.2 Implementation of Quality Rules and Reasoning for Problem Elimination Suggestions

Actually, the realisation of each quality rule is similar. The only difference lies in the solution suggestion part of the rule. Queries that only check for the existence of specific requirement information will suggest to complete the relevant missing information if they fail. In our example this would be for rules 1 (all requirements must be complete). In contrast, rule 3 (no requirement that is a negative contribution to a goal to be achieved) checks for a specific undesired requirement pattern. Thus, we do not only want to suggest to eliminate this, but more important, provide concrete suggestions what to do instead. For this rule the following options for eliminating the quality problem are:

1. If the requirement is optional, exclude it from the requirement configuration.
2. If an alternative requirement with no negative contribution to a goal is available instead, use this.
3. Revise the goal contribution relationship.
4. Modify the requirement.

These options include suggestions for solutions that need no further reasoning or any computation ( (3) and (4)). But the first two suggestions require a further inspection of the requirement and its relationships. First, we need to check whether it is optional. If this is the case, it might be a good idea to exclude it from the requirements configuration. Secondly, we check if an alternative requirement is specified. If so, we further have to inspect the goal contribution of the alternative. Only if it is not negative, we can recommend to use this requirement instead.

All of the previous listings above illustrate how to realise these steps.
5.3 Measuring the Quality of SRS

As explained in section 3 we base our quality measures on quality rules and the completeness and consistency of requirement knowledge. The previous sections illustrated how to identify missing information. In the following, we use the second solution approach to demonstrate how to apply NBox Reasoning for quality calculations. For space reasons we will only demonstrate how to measure the completeness metric described in section 3. The remaining five metrics are quite similar.

**Internal Completeness** First, we need to query for all complete requirements, these are requirements with a source, an author, a goal, a use-case, a test-case and a risk specified. Additionally, we need the number of all requirements in the SRS.

```java
Listing 1.7: SPARQL Queries for Measuring Internal Completeness
2 String queryNumberOfRequirements = "SELECT ?r WHERE {?r a ro:Requirement}";
```

Finally, we have to execute these queries and count the number of individuals. This is accomplished by the following code snippet.

```java
Listing 1.8: SPARQL Queries for Measuring Internal Completeness
1 public static int computeCompleteness(){
2     Set<String> finalSet = new HashSet<String>();
3     Query qV = QueryFactory.create(getQueryAccessor().
4         getQueryCompleteRequirements());
5     QueryExecution qe = QueryExecutionFactory.create(qV , ontModel ) ;
6     ResultSet rs = qe.execSelect();
7     ResultSetRewindable rsrw = ResultSetFactory.copyResults(rs);
8     double numberOfComplete = rsrw.size();
9     if (numberOfComplete != 0) {
10         double completeness = (numberOfComplete/getNumberOfRequirements());
11         int result = new Double(Math.round(completeness*100)).intValue();
12         return result;
13     } else return 0; }
```

Lines 4 to 6 execute the SPARQL query `queryCompleteRequirements`. The results (all complete requirements) are returned as a ResultSet. The method `getNumberOfRequirements()` only returns the SPARQL query as String. Since we only need to know the number, we use the `size()` method of ResultSetRewindable to count all returned individuals (lines 7 and 8). The percentage of completeness is measured in lines 11 and 12. In line 11 the query from listing 1.7 is executed in the method `getNumberOfRequirements()`.
Cumulative Quality of SRS  Finally, the code below demonstrates our approach for measuring the cumulative quality. Therefore, we use the single quality metrics and multiply them with the appropriate weight (line 3).

```
Listing 1.9: Measuring the Cumulative Quality of the SRS
1  public static double computeQuality()
2  {
3      double quality =
4          (((1.0*correctness)+(0.7*verifiability)+(1*consistency)+(0.6*traceability)
5           +(0.5*uncriticality)+(1*completeness)) / (1.0+0.7+1+0.6+0.5+1));
6      int result = new Double(Math.round(quality*100)).intValue();
7      System.out.println("The quality of the SRS measures: " + result + \\
8                ");
9      return result;
10  }
```

6 Conclusion and Outlook

The quality of a software requirement specification is of main importance for high-grade software products and the success of the software development process. Facilitated by ontology reasoning with local closed world we allow to identify and correct incomplete and inconsistent information and to satisfy a predefined set of quality rules. Furthermore, we developed quality metrics to calculate the quality of the SRS. A generalization of our approach lead to guidelines to adapt quality rules and metrics for any other domain. All these features have been integrated into a Eclipse Plugin and can be accessed through a yet rudimentary java user interface.

We have extended our approach proposed in [7] and have performed another in-depth evaluation using an extant set of project requirements. This evaluation has shown that the approach is capable of dealing with a reasonably complex set of requirements from a real-world problem, and can quickly identify quality flaws of a SRS. The performance is such that it can be integrated into the requirements engineering workflow without becoming a burden on the requirements engineer. Further work in this area concentrates on guidance of the requirements engineering process and an automatic configuration of requirements with respect to predefined user priorities (e.g. low risk, best goal satisfaction). The requirements ontology can be integrated with other ontological models which cover other aspects of the software development process.

References


15. TrOWL. http://trowl.eu/.