PROCESSING, CHECKING, AND MODELING OF TEXTUAL REQUIREMENTS SPECIFICATIONS

by

David Šenkýř

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics
Department of Software Engineering

Prague, November 2023

Supervisor:
prof. Dr. Ing. Petr Kroha, CSc.
Department of Software Engineering
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Copyright © 2023 David Šenkýř

Abstract

The quality of requirements engineering plays an important role in the whole development life cycle of every software project – because the other phases depend on it. The key result of the requirements engineering phase is a requirements specification. Writing requirements specifications in natural language is a common practice. However, textual requirements specifications are, unfortunately, prone to several inaccuracies, such as *ambiguity*, *inconsistency*, and *incompleteness*.

In this thesis, we map the state-of-the-art methods of grammatical inspection to identify patterns in requirements specification written in the textual form. On their basis, we are able to extract the information from the text (*text mining*) and generate warning messages targeting suspicious text formulations. We also discuss the integration of semantic networks, such as *ConceptNet* and *BabelNet*, for text defect detection. Additionally, the semantic networks are helpful in finding missing rules that are not evident from the current state of the requirements – we call them *default consistency rules*.

We present our idea of functional requirements quality measurement and a method of how analysts could iteratively improve the textual requirements.

As a software artifact verifying our approach, at the end of this thesis, we demonstrate features of our created CASE tool called TEMOS. We present ideas on how such a tool could be implemented, too. TEMOS is able to generate fragments of the UML class model from textual requirements specification, and it also helps the user detect in the text signs of incompleteness or inaccuracies that may cause ambiguity and inconsistency.

The main contributions of this dissertation thesis are as follows.

1. Identify types of problems introduced by *ambiguity*, *incompleteness*, and *inconsistency*.
2. Design of methods identifying such problems in textual requirements specifications.
3. Propose how an analyst could improve the textual requirements according to our defined quality measurement metric.

Keywords:
requirements specification, requirements engineering, natural language processing, text mining, grammatical inspection, ambiguity, incompleteness, inconsistency, quality measurement, domain model, semantic networks.

Abstrakt

Kvalita zpracování dílčích úkolů disciplíny zvané requirements engineering (ve volném překladu *inženýrství požadavků*) hraje důležitou roli pro celý životní cyklus vývoje softwarového projektu – protože to je fáze, na které ostatní fáze závisejí. Klíčovým výstupem fáze requirements engineering je specifikace požadavků, jejichž formulace v přirozeném jazyce je běžnou praxí. Textově vyjádřené požadavky jsou však náchylné k řadě nepřesností jako je *víceznačnost*, *nekonzistence* či *neúplnost*.

V této dizertační práci mapujeme aktuální stav metod gramatické inspekce za účelem identifikování vzorů v textových specifikacích požadavků. Na jejich základě jsme schopni extrahovat informace z textu (disciplína zvaná *text mining*, ve volném překladu *dolování dat z textu*) a generovat varování ohledně podezřelých textových formulací. Zároveň diskutujeme integraci sémantických sítí, jako je *ConceptNet* či *BabelNet*, za účelem detekce nepřesností v textu.

Dále popisujeme naši metriku měření kvality funkčních požadavků a navrhujeme, jak by analytici mohli iterativním přístupem vylepšit textové požadavky.

V závěru práce demonstrujeme funkcionality námi vytvořeného CASE nástroje TEMOS, který vznikl jako softwarový artefakt ověřující námi navržené postupy. Zároveň představujeme nápady, jak by podobný nástroj mohl být implementován. TEMOS je schopný generovat fragmenty UML diagramu tříd z textové specifikace požadavků a zároveň pomáhá uživateli detektovat neúplné požadavky či nepřesnosti v textu, které mohou ve výsledku způsobit víceznačnost či nekonzistenci v pochopení popisu požadavků.

Hlavní přínosy této dizertační práce shrnují následující body.

1. Identifikace typů problémů způsobených *víceznačností*, *neúplností* či *nekonzistencí.*
2. Návrh metod identifikujících zmíněné problémy v textových specifikacích požadavků.
3. Návrh postupu, jak by mohl analytik zlepšit textové požadavky s ohledem na námi definovanou metriku kvality.

Klíčová slova:

specifikace požadavků, requirements engineering, zpracování přirozeného jazyka, text mining, gramatická inspekce, víceznačnost, neúplnost, nekonzistence, měření kvality, doménový model, sémantické sítě.

Acknowledgements

First, I would like to express my gratitude to my supervisor, professor Petr Kroha, who guided me throughout this academic research journey. I appreciate his encouragement, research experience, and the valuable insights he shared with me.

Special thanks go to the staff of the Department of Software Engineering, who maintained a pleasant environment for my research. I am very grateful to the department management – Dr. Michal Valenta, Dr. Alena Libánská, and Ing. Adéla Svítková – for their positive support and help with my research-connected events organization.

I also had the pleasure of being a member of the Centre for Conceptual Modelling and Implementation (CCMi). I enjoyed all the exciting discussions with my colleagues. I would especially like to thank Dr. Marek Suchánek for his continuous encouragement and help with the deployment of our TEMOS tool. Special thanks go to Dr. Marek Skotnica for the support with the text analysis experiments regarding DEMO (Design & Engineering Methodology for Organizations).

My research has also been partially supported by the Grant Agency of the Czech Technical University in Prague, specifically by the "Advanced Research in Software Engineering" grants No. SGS17/211/OHK3/3T/18 and SGS20/209/OHK3/3T/18.

∼

Above all, my greatest thanks go to my family, especially to my parents and my sister, for their infinite patience, care, and all the support and love they give me. I wish I could turn back time and share my life stories with all of them again.

Contents

Abbreviations **xvii**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Problem Statement . . . 3
1.3 Goals of the Dissertation Thesis . . . 3
1.4 Structure of the Dissertation Thesis . . . 4
1.4.1 Research Approach . . . 5

2 Research Domain **7**

2.1 Requirements Engineering . . . 8
2.1.1 Requirements Engineering Process . . . 8
2.1.2 Software/System Requirements Specification . . . 9
2.2 Natural Language Processing . . . 12
2.2.1 Natural Language Processing Approaches . . . 13
2.2.2 Natural Language Processing for Requirements Engineering . . . 13
2.3 UML Class Diagram Generation . . . 14
2.3.1 UML Class Diagram . . . 15
2.3.2 Serialization Formats . . . 17

3 Overview of Our Approach **19**

3.1 Main Algorithm . . . 20
3.2 Text Mining Process Pipeline . . . 20
3.3 The Method of Grammatical Inspection . . . 22
3.4 Internal Model (Manager) . . . 23
3.4.1 Internal Model (Manager) Constraints . . . 24
3.5 Suitable Patterns . . . 25
3.5.1 Triplet Recognition . . . 25
3.5.2 Triplet Recognition – Challenges . . . 26

3.5.3 Attributes Recognition . . . 27
3.5.4 Hierarchy Recognition . . . 29
3.6 Evaluation . . . 30
3.6.1 Data Set . . . 30
3.6.2 Textual Modeling System (TEMOS) . . . 34

4 Problem of Ambiguity in Textual Requirements Specification 35
4.1 Motivation . . . 36
4.2 Problem Statement and Related Work . . . 37
4.2.1 Ambiguity of Words . . . 37
4.2.2 Ambiguity of Sentences . . . 37
4.3 Our Approach – Patterns of Structural Ambiguity . . . 38
4.3.1 Patterns of Attachment Ambiguity . . . 38
4.3.2 Patterns of Analytical Ambiguity . . . 41
4.4 Our Approach – Glossary Construction and Synonyms Resolving . . . 43
4.5 Experiments and Results . . . 44
4.6 Problems of Semantic Sentence Ambiguity and Coreference . . . 48
4.6.1 Linguistic Approach Completed by Knowledge Base . . . 48
4.6.2 Model Approach . . . 49
4.6.3 Specific Attribute Values Distinguish the Coreference . . . 49

5 Problem of Incompleteness in Textual Requirements Specification 53
5.1 Problem Statement . . . 54
5.2 Related Work . . . 56
5.2.1 Incompleteness Detection Tools . . . 56
5.2.2 Incompleteness Confrontation . . . 57
5.2.3 Incompleteness of Scenarios . . . 58
5.2.4 Related Works – Overview . . . 59
5.3 Our Approach . . . 59
5.3.1 Group S.1 (Usual Usage of Words) . . . 60
5.3.2 Group S.2 (Acronyms Definition) . . . 61
5.3.3 Group D.1 (Semantic Knowledge) . . . 61
5.3.4 Group D.2 (Actions) . . . 62
5.3.5 Group D.3 (Model Validation) . . . 62
5.4 Our Approach – Incompleteness of Scenarios . . . 63
5.4.1 Alternative Scenarios . . . 63
5.4.2 The Algorithm . . . 64
5.4.3 Static UML Model Construction . . . 65
5.4.4 Sets of Values . . . 65
5.4.5 Patterns To Find Scenarios . . . 67
5.5 Experiments and Results . . . 70
5.5.1 Evaluation Example #1 . . . 70

5.5.2 Evaluation Example #2 72
5.5.3 Results 72

6 Problem of Inconsistency in Textual Requirements Specification 75
6.1 Problem Statement 76
6.2 Related Work 76
6.3 Sources of Inconsistency 77
6.3.1 Semantic Overlaps as Sources of Inconsistency 77
6.3.2 Inconsistency between the Text and the UML Model 78
6.4 Our Approach 79
6.4.1 Model Construction and Semantically Similar Sentences 80
6.4.2 Example – Library Information System 82
6.5 Experiments and Results 84
6.5.1 Data 84
6.5.2 Results 85

7 Problem of Default Consistency Rules in Textual Requirements Specification 89
7.1 Problem Statement 90
7.2 Case Study – Part 1: Missing Consistency Rules in Chords Generation 90
7.2.1 Chord Generation Requirements 91
7.2.2 Our Approach – Using External Context to Identify the Missing Default Consistency Rules 92
7.2.3 Construction of Pseudo-Questions 92
7.2.4 Semantic Similarity of Sentences 93
7.2.5 Semantic Enrichment of Sentences 93
7.2.6 The Process of Semantic Enrichment of Sentences 94
7.3 Case Study – Part 2: Applying Our Approach 96
7.3.1 The Missing Consistency Rule No. 1 96
7.3.2 The Missing Consistency Rule No. 2 98
7.4 Experiments and Results 99
7.4.1 Revealing Consistency Rule No. 1 100
7.4.2 Revealing Consistency Rule No. 2 101
7.4.3 Discussion 101

8 Quality Measurement 103
8.1 Problem Statement 104
8.2 Quality Measurement 105
8.3 Our Approach to Quality of Requirements 107
8.4 Experiments and Results 109

9 Created Artifact and Models Generation 113

9.1 TEMOS – Textual Modeling System 114
9.1.1 Core Features 115
9.1.2 Additional Features 116
9.1.3 Used Technologies 116
9.2 Models Generation 117
9.2.1 Generated Models – UML (Class Diagram) 117
9.2.2 Generated Models – SHACL 120
9.2.3 Generated Models – Normalized Systems 125
9.2.4 Generating Information System Prototypes 128

10 Conclusions **129**
10.1 Research Goals Revisited 129
10.2 Contributions of the Dissertation Thesis 130
10.3 Future Work 131

Bibliography **133**

Reviewed Publications of the Author Relevant to the Thesis **147**

Remaining Publications of the Author Relevant to the Thesis **151**

Selected Relevant Supervised Thesis **153**

List of Figures

1.1 Software development life cycle (SDLC). 2
1.2 Overview of the structure of this thesis. 6

2.1 Problems of textual requirements specifications. 11
2.2 Result of tokenization process. 12
2.3 Result of part-of-speech tagging. 12
2.4 Class diagram example of hotel management system. 15
2.5 Class diagram – association example. 16
2.6 Class diagram – generalization example. 16
2.7 Class diagram – aggregation example. 17
2.8 Class diagram – composition example. 17

3.1 Example of an annotated sentence. 21
3.2 Basic triplet pattern. 25
3.3 Example of the passive voice #1. 26
3.4 Example of the passive voice #2. 27
3.5 Example of a sentence with attribute candidate and the verb *have*. 27
3.6 Example of a sentence with attribute candidates. 28
3.7 Example of a sentence with an indirect subject. 29
3.8 Example of a sentence representing hierarchy of entities. 29
3.9 Example of a sentence representing attribute values recognition. 30
3.10 TEMOS pipeline. 34

4.1 Pattern #1 (prepositional phrase modifier). 39
4.2 Pattern #2 (preposition phrase). 39
4.3 Pattern #3 (relative clause). 39
4.4 Pattern #4 (subsentence attachment). 40
4.5 Pattern #5 (adverbial position (1)). 40
4.6 Pattern #6 (adverbial position (2)) – example sentence. 41
4.7 Pattern #6 (adverbial position (2)). 41

4.8 Pattern #7 (reduced restrictive relative clause (1)). 41
4.9 Pattern #8 (reduced restrictive relative clause (2)). 42
4.10 Pattern #9 (present participle vs. adjective). 42
4.11 Pattern #10 (participle). 42
4.12 Ambiguous parsing structure in Grammarly. 44
4.13 Example of coreferential ambiguity. 48
4.14 USE tool example. 51

5.1 Example of incompleteness (restaurant). 55
5.2 Noun with preposition pattern. 60
5.3 "In order to action" pattern. 60
5.4 Values conditional pattern #1. 68
5.5 Values conditional pattern #2. 68
5.6 Incomplete conditional pattern #1. 68
5.7 Incomplete conditional pattern #2. 68
5.8 Unique attribute pattern. 69
5.9 Values in enumeration – check approach. 70

6.1 The idea of inconsistency patterns. 80
6.2 Pattern #1 (pure negation). 81
6.3 Matched pattern #1 (pure negation). 81
6.4 Pattern #2 ("except"). 81
6.5 Pattern #3 (numeric predicate). 81
6.6 Pattern #4 (determiner predicate). 81
6.7 Relation actor predicate. 82
6.8 Pattern #5 (predicate and auxiliary verb). 82
6.9 Consistency rule from Example 6.2. 85

7.1 Example of a not playable chord. [21] 91
7.2 Example of a not playable chord – three tones on one string. 92
7.3 Chord definition from WordNet – the scope of the definition exceeds string instruments. 99

8.1 Quality evaluation schema. 107
8.2 Quality measurement iterations as feedback. 108
8.3 Correlation comparison of IPA and ARI. 112
8.4 Correlation comparison of EN and Q-Req. 112

9.1 TEMOS – main page. 114
9.2 Generated elements. 115
9.3 Generated warnings concerning ambiguity. 116
9.4 Generated data elements for a hotel room booking system example. 127
9.5 Booking form for expanded NS application. 128

List of Tables

2.1 Requirements specification notations. [107] 10

3.1 Data set statistics. 32

4.1 Evaluation of methods detecting ambiguity. 46

5.1 Evaluation of methods detecting incompleteness. 73

6.1 Evaluation of methods detecting inconsistency. 87

7.1 Google Web search: *chord* and *chord* + *fingering*. 101

8.1 The evaluation of the recognized issues (warnings) using Q-Req formula. 110

List of Algorithms

3.1 Our approach – main algorithm. 20

5.1 Revealing alternative scenarios. 64

7.1 The semantic enrichment of sentences. 94

Abbreviations

Requirements Engineering & Conceptual Modeling

BRS	business requirements specification
CASE	computer-aided software engineering
EMF	Eclipse Modeling Framework
FRS	functional requirements specification
MDD	model-driven development
NLP4RE	natural language processing for requirements engineering
NS	Normalized Systems
OCL	object constraint language
RDF	resource description framework
RE	requirements engineering
SDLC	software development life cycle
SHACL	shapes constraint language
SRS	software/system requirements specification
UML	unified modeling language
USE	UML-based specification environment

Text Mining

POS	part-of-speech
NLP	natural language processing

Miscellaneous Abbreviations

API	application programming interface
ARI	automated readability index
GFI	Gunning fog index
GUI	graphical user interface
XMI	XML metadata interchange
XML	extensible markup language

Chapter 1

Introduction

The idea of requirements specification is not new. The famous Italian painter Raffael (1483–1520) produced his paintings to the orders of wealthy customers. They wanted to know in advance what they would pay for. Rafael described the picture in words, for example, "In the oil painting on canvas with dimensions according to the His Highness's wishes, there will be His Highness like a knight in armor on a white horse with a sword and spear fighting a dragon" and added a sketch (drawing) of the painting. The customer could opt, for example, for a black horse and a helmet with a plume. Then they wrote a contract, and the image had to match its content. After five hundred years, we follow (more or less) the same practice in software engineering projects. The description of the software product to be constructed is called software requirements specification.

1.1 Motivation

The significant phase of the *software development life cycle* (SDLC[1]) is undoubtedly the elicitation, the investigation, and the processing of a *requirements specification*. Actually, by well-known standard phases of SDLC illustrated in Fig. 1.1, the mapping of the requirements specification is the first one or one of the first outputs from the initial investigation of a software system utilization. The requirements specification is also the essential input for the subsequent development steps – such as modeling individual parts of a system. The quality of the outputs of these steps is, of course, dependent on the quality of the input requirements because they are mandatory prerequisites. Although agile projects include more iterations more or less similar to the SDLC cycle, the precise understanding of requirements in each iteration is still crucial [85].

According to Tichy et al. [80], the quality of the input requirements has the most expensive impact. As Kof stated in his paper [75] – *"Requirements Engineering is the Achilles heel of the whole software development process."*

[1]In some literature also used as the *system development life cycle.*

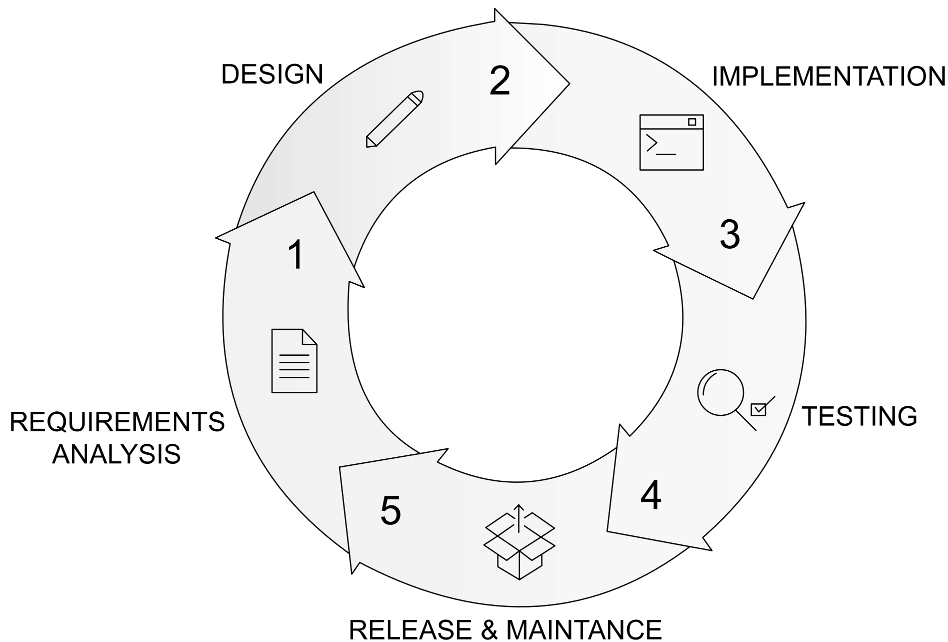


Figure 1.1: Software development life cycle (SDLC).

Textual formulated requirements specification is necessary as a base of communication between a customer, a user, a domain expert, and an analyst. The majority of textual requirements specification – compared to other forms, such as graphical notations (i.e., diagrams) or mathematical specifications – is observed in the market research surveys.

The market research [82] of 151 questionnaires from the early 2000s states that 79 % of all specifications are written in common natural language without any structure or formalism. A decade later, the majority of respondents (61 %) still report that their requirements are expressed in terms of natural language [72]. The situation remains unchanged even in recent research. The NaPiRE (Naming the Pain in Requirements Engineering) initiative [127] confirms that the most frequent way to document requirements is free-form textual structured requirements lists. The majority of natural language sentences usage is declared in [44], too.

The textual formulation of requirements in natural language is also necessary due to a contract with the clients. The contract is then the primary relevant source that can be assessed in the event of a legal case. Unfortunately, requirements texts typically suffer from *ambiguity*, *incompleteness*, and *inconsistency*. Usually, the reason is that writing requirements is a cooperative work of several people who are often distributed in various places. This is a source of inaccuracies and misleading descriptions. Many words and statements may have multiple meanings (*ambiguity*), text can obtain contradictions (*inconsistency*), and specifications of some features can be omitted (*incompleteness*).

Given the severity of requirements specification and the new possibilities of computer-aided support for *natural language processing*, there is a motivation for developing CASE

tools supporting requirements engineering. Tools that assist in mapping parts of textual requirements specification to corresponding fragments of some model.

In this dissertation thesis, we present the state-of-the-art of *static model generation* (primary UML class diagram) and techniques for handling mentioned issues of a textual formulation. We also present our tool TEMOS that we have developed to process *textual formulated requirements specification.*

1.2 Problem Statement

In our research, we address the problem domain called *NLP4RE* – Natural Language Processing for Requirements Engineering. This is an area of research and development that seeks to apply natural language processing (NLP) techniques, tools, and resources to the requirements engineering (RE) process, to support human analysts to carry out various linguistic analysis tasks on textual requirements documents. [131]

> *"Software requirements sit in a tricky zone between business and technical thinking."*
>
> — James Billson, CEO at Primary.app [14]

The Most Surprising Software Project Failure Statistics And Trends in 2023 [50] collects data from various sources such as *The Standish Group's Chaos Report*, *Project Smart UK*, *InfoQ* articles, and *PMI of the Profession* reports. They state that:

- 32 % of software project failures are due to poor requirements management,
- 23 % of software project failures can be traced back to poor communication between stakeholders.

Following the beforehand mentioned statistics and the NLP4RE problem domain, we define the problem statement as *processing, checking, and modeling of textual requirements specification.* Accordingly, we formulate the objectives of this dissertation thesis in the following section.

1.3 Goals of the Dissertation Thesis

In general, our motivation is to support analysts, stakeholders, developers, or anyone else who creates, updates, or just reads the textual requirements specifications. Therefore, the main goal on the abstract level should be stated as *to improve the text of textual specification.* Because of that, we define the following more concrete goals.

1. **G1.** Identify the type of problems introduced by:
   a) **G1.A** ambiguity,
   b) **G1.B** incompleteness, and
   c) **G1.C** inconsistency.
2. **G2.** Propose algorithms on how to identify such problems in textual requirements.
3. **G3.** Propose a method for how an analyst could improve the text.

1.4 Structure of the Dissertation Thesis

This thesis is organized into ten chapters as follows:

Chapter 1 (Introduction) describes the motivation behind our efforts together with our goals. There is also a list of contributions of this dissertation thesis.

Chapter 2 (Research Domain) introduces the reader to the necessary theoretical background relevant to our research domain – requirements engineering and natural language processing.

Chapter 3 (Overview of Our Approach) explains our approach and introduces our internal model representing the parsed text of requirements.

Chapters 4, 5, 6, and 7 address the issues defined in goal **G1** – *ambiguity*, *incompleteness*, and *inconsistency*. During the investigation, we faced facts and rules that were not part of the requirements specification. We call them *default consistency rules*, and we devote a separate chapter to them.

Chapter 8 (Quality Measurement) summarizes the results of investigated problems from the previous chapters and provides the quality measurement formula.

Chapter 9 (Created Artifact and Models Generation) describes our system TEMOS as a created software artifact implementing the proposed methods from Chapters 3–8. This chapter also addresses model generation and model export in various formats.

Chapter 10 (Conclusions) summarizes the results of our research, suggests possible topics for further research, and concludes the thesis.

1.4.1 Research Approach

To achieve the goals, we proceeded in our research activities as shown in Fig. 1.2 that gives an overview of the structure of this thesis and the connection to the defined goals.

First, we review the necessary background of requirements engineering and natural language processing support, and we present the domain in the next chapter. Then, we prepare a structure of our internal model representing the parsed text of requirements. This structure helps us support goal **G2** – the idea of a model representation makes a step toward checking problems defined in goal **G1** on the semantic level. We present this structure in Chapter 3.

To address each problem of *ambiguity*, *incompleteness*, and *inconsistency*, we investigate them in separate chapters where we identify the typical manifestations of the problem (goal **G1**), and we propose algorithms to identify them in the text (goal **G2**). With each proposed solution, we extend our created software artifact – system TEMOS. This system is presented as a solution to support analysts when they process the text of requirements (goal **G3**).

Our approach to overall text improvement (goal **G3**) is to use the results of investigated problems detection and our proposed quality measurement metric as presented in Chapter 8. The aforementioned system TEMOS (also supporting goal **G3**) is presented in Chapter 9.

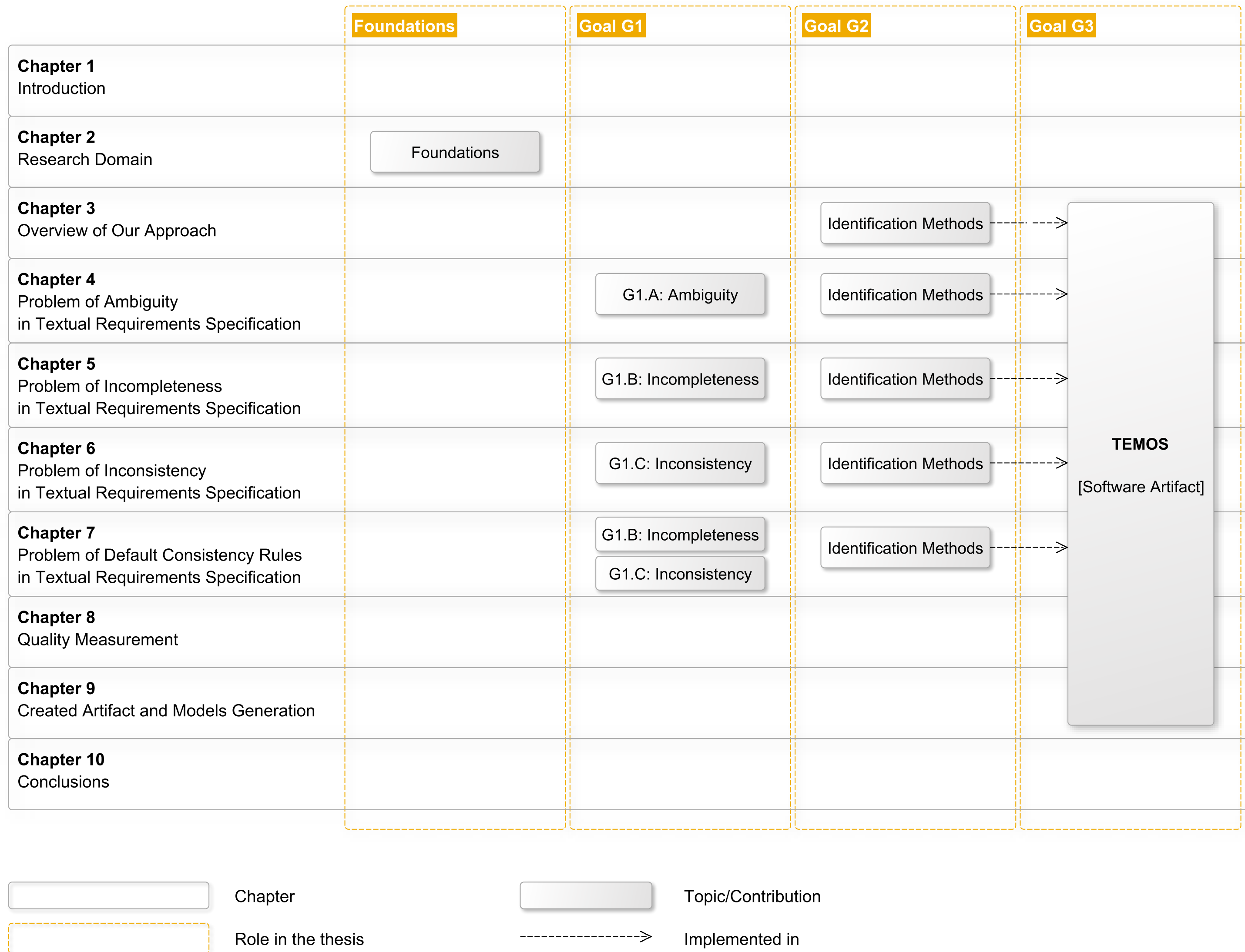


Figure 1.2: Overview of the structure of this thesis.

Chapter 2

Research Domain

This chapter reflects our publication:

- Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies.* SciTePress, Madeira, 2018. [A.10]

In this section, we clarify the terms and methodologies of our research domain. First, we introduce requirements engineering. Next, we present the basic approach of natural language processing. We introduce the research area connecting requirements engineering and natural language processing called NLP4RE – natural language processing for requirements engineering. Last, we present propositions about UML class diagram generation.

2.1 Requirements Engineering

A central concept of our research domain is represented by a *requirement*. We follow the definitions provided by *IEEE*, and we appreciate discussion of the terms presented in a book called *Requirements Engineering* [28] by Jeremy Dick, Elizabeth Hull, and Ken Jackson.

Definition 2.1.1. A *requirement* is a statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality assurance guidelines). [64]

A valid note concerning a requirement is that the requirement states what is required, not how the requirement should be met. [65] To tackle the requirements engineering process, let us first define a stakeholder.

Definition 2.1.2. A *stakeholder* is an individual, group of people, organisation or other entity that has a direct or indirect interest (or stake) in a system to be developed. [28]

Definition 2.1.3. The process called *requirements engineering* is the initial process by which it comes into contact an analyst and a client (a stakeholder) to clarify the client's expectations of the future software.

Definition 2.1.4. The (software/system) *requirements specification* (abbreviated *SRS*) verified by a client (a stakeholder) is then the main output of the requirements engineering process.

2.1.1 Requirements Engineering Process

The requirements engineering process primary consist of the following phases [108]:

- **elicitation** – meetings and appointments, observations of users, etc.,
- **analysis** – thinking and inventing, discussions, notes, etc.,
- **specification** – writing documents, using agreed notation, etc.,

- **verification** – other meetings, reading documents, presenting prototypes, clarifying the scope of functionality, etc.

These phases can be repeated multiple times with various people from various departments.

2.1.2 Software/System Requirements Specification

The software/system requirements specification covers the scope of the future software – which should be characterized, at least, by the following categories [108]:

- **functional requirements** – requirements related to software functionality like the workflow[3] of tasks and activities that will be supported,
- **interface requirements** – user interface design, software/hardware integration requirements, etc.,
- **non-functional requirements** – properties of systems as a whole like performance (e.g. response time), accessibility and availability, extensibility and scalability, security, etc.,
- **other requirements** – legislative, multilingualism, etc.

From the formal point of view, there exist standards like *ISO/IEC/IEEE 29148-2018*[4] [66], methodologies like *SWEBOK Guide* [105] or *Volere Requirements Specification Template* [55] (that is translated in various languages), as well as CASE tools that support better requirements organization or even manual assignment of requirements with parts of the model like *Enterprise Architect* [118].

The *SWEBOK Guide* (The Guide to the Software Engineering Body of Knowledge), at the time of writing this thesis in version 3, published by IEEE, includes a chapter dedicated to previously mentioned requirement engineering phases with guidelines and best practices.

However, following the mentioned market research in Chapter 1, the most widely used approach is to write requests in natural language as non-structured text. And these are the specifications we are interested in. The other approaches to capture requirements are categorized, e.g., in [107]. We present a summary in the following adopted Table 2.1.

[3] The set of inputs, the behavior, and the set of outputs.

[4] ISO/IEC/IEEE 29148-2018: Systems and software engineering – Life cycle processes – Requirements engineering. The successor of IEEE 830-1998: Recommended Practice for Software Requirements Specifications.

Table 2.1: Requirements specification notations. [107]

Notation	Description
natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
gaphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Many stakeholders who contract software projects, and IT projects in general, obviously operate outside the IT sector. They are experts in their business domain. In these cases, the natural language is surely the clear choice for describing the expected system functionality and requirements. Otherwise, even if the client knows more formal methods of requirements formulation like diagrams, models, etc., requirements formulation in the natural language is necessary because of a contract. The contract is then the primary relevant source that can be assessed in a legal case.

The advantage of using natural language is that the requirements specification can be interpreted both by a customer (or other stakeholders) and by an analyst. However, the freedom without any formal restriction makes them prone to a number of inaccuracies and incomplete expressions. Previous customer nescience of the formal methods of requirements formulation is now substituted by analyst nescience of the customer's business domain. What is natural for the domain expert from the customer team may not be fully evident for the analyst – also a domain expert but in a different domain.

Writing requirements may also be a cooperative work of several people. When the contractor is a company, it can be assumed that it is almost a rule. The cooperative work is another source of inaccuracies and misleading descriptions. For example, describing a

term in the text by several synonyms may – in the case of not well-known terms or domain-specific terms – result in a situation where the analyst denotes synonyms as various terms.

Maintaining requirements specifications written in natural language complete and straightforward is a difficult task with a high probability of introducing new inaccuracies, as a schema adapted from professor Easterbrook's presentation [30] in Fig. 2.1 illustrates.

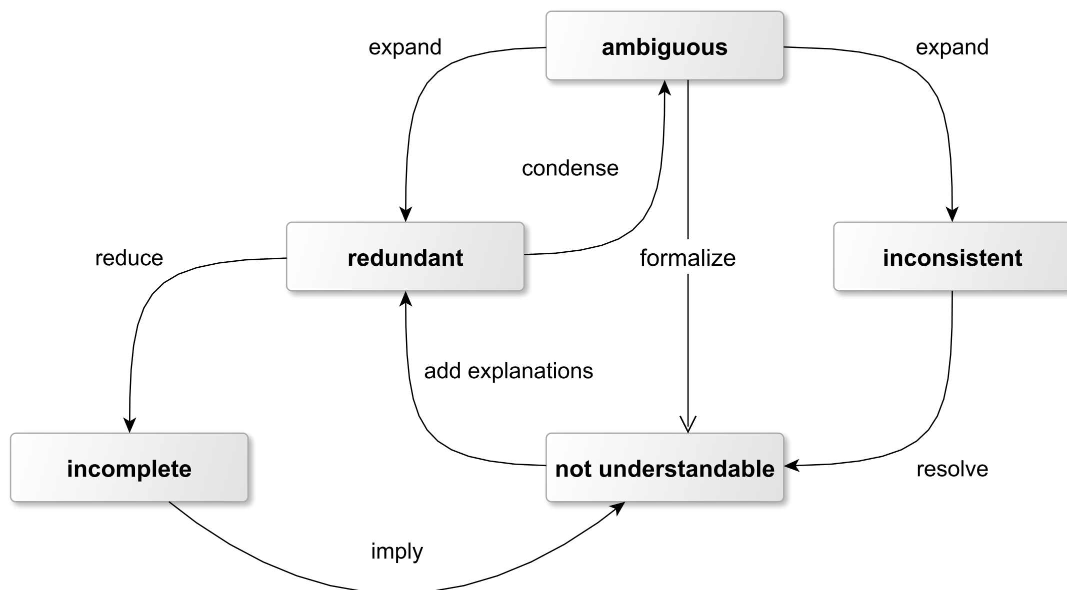


Figure 2.1: Problems of textual requirements specifications.

Software requirements specification (SRS) can be a follow-up document to the business vision stated in *business requirements specification* (BRS). According to IEEE standard [66], the *business requirements specification* describes the organization's motivation for why the system is being developed or changed, defines processes and policies/rules under which the system is used and documents the top-level requirements from the stakeholder perspective including expressing needs of users/operators/maintainers as derived from the context of use in a specific, precise and unambiguous manner.

As mentioned at the beginning of this section, *software requirements specification* (SRS) can cover different kinds of requirements. The document extracting only the core functional requirements is then called *functional requirements specification* (FRS).

2.2 Natural Language Processing

Computational processing of natural language requires the collaboration of engineers and especially linguistic experts. A multidisciplinary field devoted to this domain is called *computational linguistics*. Its origins date back to the 1950s [62], when there was the first mention of computer-controlled translations of text from one language to another. In recent years, there were created *Natural Language Processing* (NLP) frameworks that can be used by developers. The increasing evolution of these frameworks offers opportunities in the fields of *information retrieval*, *text mining*, *question answering*, *speech recognition*, etc.

These frameworks typically provide some of the following operations:

- **tokenization** – usually the first step when a framework parses the input text (the set of characters) into a sequence of tokens; a single token (a unit carrying significance) represents a word or a special character like an interpunction, etc.; whitespace characters (like spaces) don't represent tokens, but they are control characters for token recognition; the result of the tokenization process is shown in Fig. 2.2 – the parts of the text bounded by the top curve with the `T` character represent individual tokens (as mentioned before, the dot character is also a separate token),

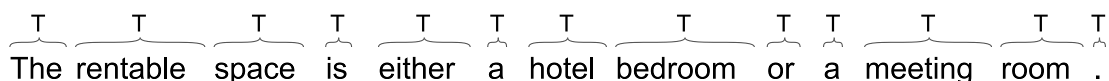


Figure 2.2: Result of tokenization process.

- **sentence segmentation** – based on the tokenization, tokens are clustered into sentences,

- **part-of-speech tagging** – annotates every token with the part-of-speech (POS) tag – such as a *noun* (`NN`), an *adjective* (`JJ`), a *verb* (`VB`), *3rd person singular present verb* (`VBZ`), etc.; in our examples, we use *Penn Treebank part-of-speech tagset* [121] that is adapted in multiple NLP frameworks mentioned below; an example result of this process is shown in Fig. 2.3,

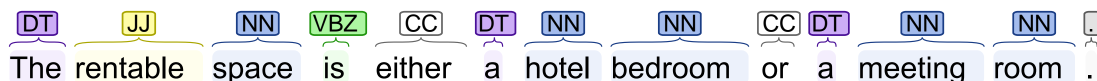


Figure 2.3: Result of part-of-speech tagging.

- **lemmatization** – a process of generation base forms (*lemmas*) for every token,

- **dependencies recognition** – searches for relationships between words (*nominal subject(s)* of a verb, *dependency object(s)* of a verb, etc.) based on the grammatical structure of a sentence,

- **coreferencies recognition** – represents a search for words which pronouns refer to.

As an example of NLP frameworks, from our experience, we can mention *Stanford CoreNLP* framework written in Java or *spaCy* and *NLTK* frameworks written in Python. One can find other frameworks in [69] and [100].

2.2.1 Natural Language Processing Approaches

In [75] and [74], Kof categorized NLP approaches into three groups as follows.

- The first is related to lexical methods – methods that do not rely on basic NLP approaches such as part-of-speech tagging or other parsing. These methods perceive text as a sequence of characters and look for terms (subsequences) that occur repetitively.

- Syntactical methods typify the second group. These methods use part-of-speech tagging and look for specific sentence constructions. Based on this, they are able to distinguish objects and relationships.

- The last group represents semantic methods – methods that interpret each sentence as a logical formula or a partial model. The goal is then to look for predefined patterns or structures.

Moreover, semantic methods should be supported by predefined patterns or structures created by humans or via machine learning [120].

2.2.2 Natural Language Processing for Requirements Engineering

As introduced in Chapter 1, *Natural Language Processing for Requirements Engineering* (NLP4RE) is an area of research and development that seeks to apply natural language processing (NLP) techniques (e.g., part-of-speech tagging), tools (e.g., NLP frameworks such as spaCy, NLTK, or Standford NLP), and resources (e.g., ontologies such as dictionaries and corpus [18]) to the requirements engineering (RE) process. [131]

Applying NLP techniques to analyze market news or posts about elections on social media could benefit from the pre-trained models for a specific domain with more or less expected dictionary. Applying NLP techniques to requirements engineering documents is more challenging because, in this case, NLP techniques should be open to multiple domains – medicine, finance, law, etc. Therefore, we see the usage of online ontologies (such as dictionaries) as a crucial discipline here.

NLP4RE is also a series of research workshops[1] organized as a part of the REFSQ conference. A mapping study of NLP4RE research area is provided in [131].

2.3 UML Class Diagram Generation

A (graphically represented) *model* can be considered as an interpretation of the *requirements*. It is common practice that the model precedes the system implementation. So, it becomes an intermediary artifact between the software analysis and the software implementation. Moreover, high-level graphical visualization of a model can be helpful when analysts clarify the requirements understanding with stakeholders even if the stakeholders do not know the details of the process used to create the model or the full expressiveness of the model.

As we describe in the next chapter (Chapter 3 – Overview of Our Approach), we process textual requirements and generate a model from them. The model we use benefits from *UML class diagram* concepts. Additionally, we are able to generate the *UML class diagram* from our model. Therefore, we present concepts of the *UML class diagram* we use in the next section. The *UML class diagram* generation is regularly considered in the NLP4RE research area as presented in the survey [1]. An overview of tools and techniques of UML diagram generation since the first attempts in the early 1980s is presented in [12]. The authors focused on the differences between manual, semi-automatic, and automatic processing of the tools.

The *UML* is an abbreviation of *Unified Modeling Language* – the language designed for visual modeling (primarily of *object-oriented software systems*). The UML was accepted in 1997 by OMG[2] as the first open, industry standard object-oriented visual modeling language. The current latest version of UML specification is 2.5.1, published in 2017. The UML specification in version 2.4.1 is proposed as standard ISO/IEC 19505 [70]. These days, the UML is de facto standard No. 1 in the software development life cycle, and it is widely supported by CASE tools.

The adjective *unified* refers to various diagrams throughout the entire development cycle, independent of an application domain, a platform, and a programming language. That is why many software design patterns are expressed using the UML [6].

There is also a development approach called *Model-Driven Development* (*MDD*) that is based on generating prototypes from the models – users can quickly get an idea of the system being developed.

Although, as mentioned, the UML provides various diagrams, we primarily focus on the UML class diagram in the following chapters because the UML class diagram is the most beneficial structure for us.

[1] `https://nlp4re.github.io/2023`

[2] The Object Management Group – international not-for-profit technology standards consortium, founded in 1989 [90].

2.3.1 UML Class Diagram

The UML diagrams are divided into categories. The UML class diagram is part of the structure diagrams category [91]. As the name suggests, the UML class diagram deals with units called *classes*. One of the key concepts of this diagram type is to map *relationships* between classes.

It is not our goal to provide a summary of all supported concepts of the UML class diagram here. The notation is fully described in the specification [91], and practical examples can be found in [6]. Here, we just want to recall the concepts that are useful for our approach so we can reference them when we talk about (UML) model generation.

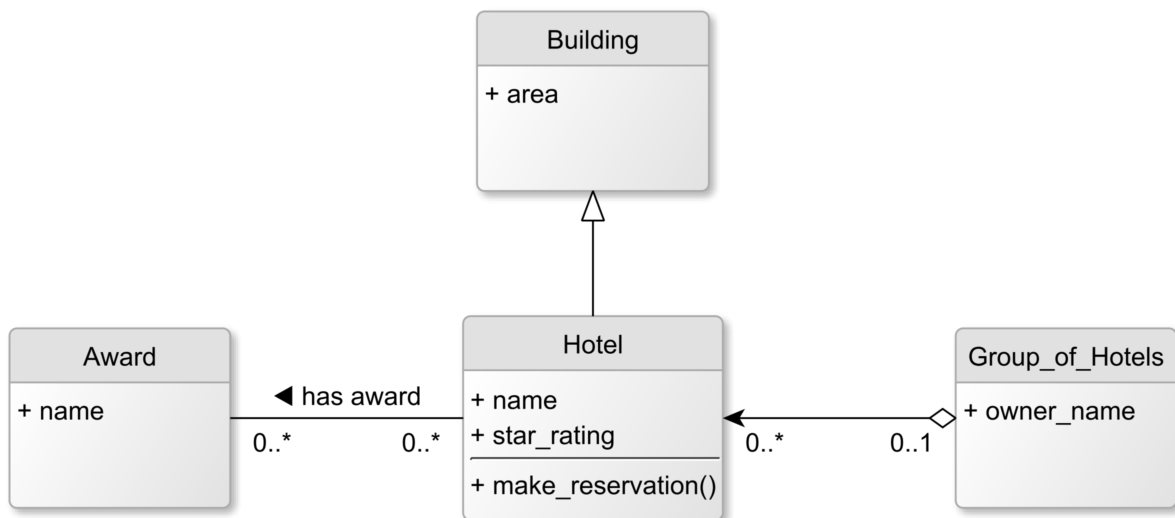


Figure 2.4: Class diagram example of hotel management system.

2.3.1.1 Concept of Classes

According to [101], a *class* is a descriptor for a set of objects with similar structure, behavior, and relationships.

The example diagram in Fig. 2.4 describes a simple model (very far from the real design) that contains four classes representing a *building*, a *hotel*, an *award*, and a *group of hotels*. For our purposes, we limit the class notation to the class name (e.g., *building*), a set of attributes (e.g., *area* (it could be defined in square meters) of a *building*), and a set of operations (e.g., *make a reservation* in a *hotel*).

Every class should have some attributes. A class without attributes is suspicious of incompleteness (as we describe in Chapter 5) – some part of the description of such a class is probably missing.

2.3.1.2 Concept of Relationships

Relationships are semantic connections between UML elements. The basic relationship between classes is called an *association* [6]. The association, among other properties, should have a name (should be a verb phrase) representing the semantic meaning of the connection between associated classes. For our purposes of the generated model check, it is also important to note that the association can have multiplicity stated for both classes that are part of the association.

There are many other relationship variants in UML, and some of them also distinguish small semantic details. For our purposes, let us recall the graphical notation of the following relationships:

- basic **association** – with a name and multiplicities; the name can have a navigation arrow to specify the direction in which the association name should be read; also, unidirectional associations can have a single arrow on one of the sides to determine which class will hold the reference – in the example in Fig. 2.5, `Class A` holds the reference to `Class B`, but not vice versa,

Figure 2.5: Class diagram – association example.

- **generalization** relationship – represents a concept called inheritance – in the sense “it is a kind of”; following the example in Fig. 2.6, `Class A` represent a *specialization* of `Class B` and `Class B` represent a *generalization* of `Class A`,

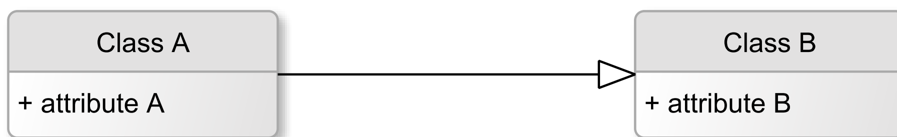


Figure 2.6: Class diagram – generalization example.

- **aggregation** – represents a more semantic detail in comparison to the basic association; aggregation is a whole-part relationship where the whole (the aggregate) uses services of the part(s) in the sense that the whole represents the controlling part of the relationship and the part(s) service requests of the whole; the key point here is that the part(s) may exist without the whole,

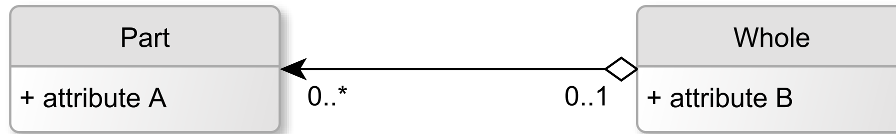


Figure 2.7: Class diagram – aggregation example.

- **composition** – is an "stronger" alternative to aggregation where the part(s) cannot exist without the whole, and the whole has sole responsibility for managing its parts.

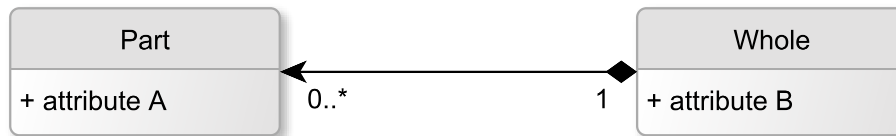


Figure 2.8: Class diagram – composition example.

The example diagram in Fig. 2.4 contains three relationships. The first one (*generalization*) between the `Building` class and the `Hotel` class means that every *hotel* is also a *building*. Therefore, every `Hotel` class also has the `area` property inherited from the `Building` class.

The second relationship is an example of *aggregation*, and in this context, it means that a *hotel* could be part of some business *group of hotels*. However, single hotels can still exist without the need to be a part of any group.

The relationship between the `Award` class and the `Hotel` class is the *bidirectional association*. The same *award* can be assigned to multiple *hotels*, and one *hotel* can be awarded by multiple *awards*.

2.3.2 Serialization Formats

The exchange of UML models between different applications is possible via serialization in XMI – the *XML metadata interchange* format. The XMI format is a standard of the Object Management Group [92].

Java projects can benefit from *Eclipse Modeling Framework* (EMF), which includes a custom meta-model called *Ecore* for describing models. EMF uses serialization in XMI, too. [53]

We use these two options as the output of our UML class diagram generation, as presented in Chapter 9 (Created Artifact and Models Generation).

Chapter 3

Overview of Our Approach

This chapter reflects our publications:

- Šenkýř, D.; Suchánek M.; Kroha, P.; Mannaert, H.; Pergl, R. Expanding Normalized Systems from textual domain descriptions using TEMOS. In: *Journal of Intelligent Information Systems*. Springer, 2022. [A.2]
- Šenkýř, D. SHACL Shapes Generation from Textual Documents. In: *Enterprise and Organizational Modeling and Simulation*. Springer, Cham, 2019. [A.7]
- Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies*. SciTePress, Madeira, 2018. [A.10]

In this chapter, we discuss the text processing methods that we use when solving all presented problems described separately in the following chapters. Next, we define our internal model that represents the parsed text of requirements. Finally, we discuss our evaluation approach.

3.1 Main Algorithm

First, we introduce our approach in the very abstract steps in Algorithm 3.1. At the beginning, we have textual requirements representing functional requirements specification document – let us look at it as the input to the algorithm. The goal is to transform the text into a domain model representation and to generate warnings and questions for users – let us look at it as the output of the algorithm.

The input text is processed by the text mining process pipeline. Once this step is complete, we use a method of grammatical inspection and our defined patterns to extract a domain model representation. Using the grammatical inspection of individual sentences and a generated domain model, we can warn the users about suspicious requirements formulations.

Algorithm 3.1: Our approach – main algorithm.

1. text mining process pipeline
2. grammatical inspection
3. domain model extraction
4. model and text requirements processing and warnings/questions generation

In this chapter, we present an overview of the first three steps of Algorithm 3.1 because they are more or less shared concerning all the problems we investigate. The fourth step is specific for each problem (*ambiguity*, *incompleteness*, and *inconsistency*) detection separately. According to the categorization of NLP approaches in Section 2.2.1, we combine both *syntactic* (text mining process pipeline) and *semantic* (domain model extraction and model quality processing) methods.

After the fourth step, the stakeholders could update the text of requirements, and the process represented by Algorithm 3.1 could be repeated. We discuss this approach regarding text quality improvements in Chapter 8.

3.2 Text Mining Process Pipeline

Let us consider the whole input text as a collection of sentences. A sentence is a couple $S = (T, D)$ where T is an ordered set of tokens t_i and D is a set of dependencies. This structure is created using a quite typical pipeline (already introduced in Section 2.2) of the text mining process that includes these phases:

tokenization identifies tokens t_i where a token is a word or interpunction,

sentence segmentation composes the sentence S based on the tokens representing interpunction,

part-of-speech tagging resolves the part-of-speech tag (e.g., `VBZ` – *3rd person singular present verb*) and part-of-speech tag category (e.g., `VERB`) of each token t_i,

lemmatization identifies the word's lemma (sometimes also called dictionary form),

dependency recognition constructs set D, where the dependencies, i.e., relations between tokens, are mapped,

co-reference recognition links tokens represented by a pronoun to their original referent (a person or thing typically represented by a (proper) noun).

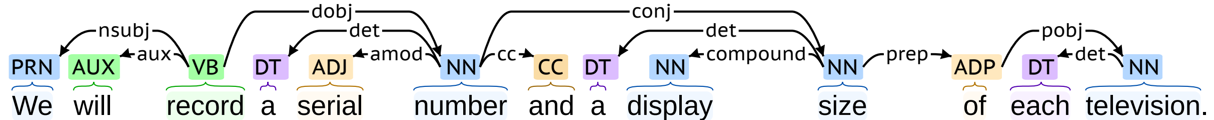


Figure 3.1: Example of an annotated sentence.

A result of this pipeline is the annotated original text of requirements that we use as a source for the method of grammatical inspection. An example of the annotated sentence is in Fig. 3.1. For our purpose, each token t_i has the following properties:

text the original text of the token presented in a sentence,

lemma a base (dictionary) form of the token text,

part-of-speech tag an assigned part-of-speech tag including details such as tense, singular/plural form indication, 3rd person indication, etc.; one can find a list of tags in [121],

part-of-speech category an assigned part-of-speech category that groups multiple tags into one category, e.g., different kinds of verbs (`VB`, `VBZ`, or `VBD`) into one category called `VERB`; one can find a list of categories in [125],

co-reference link optional co-reference link to another token.

A dependency d_j is an asymmetric binary relation between two different tokens t_1 and t_2 having a type such as *nominal subject*, *direct object*, etc. In our examples, we use types

provided by spaCy trained models for English[1]. It is a subset of Universal Dependency Relations for English [124].

To shortly discuss the structure of the example in Fig. 3.1, as humans we identify a subject (*we*), a predicate (*to record*), and two direct objects[2] (*a serial number* and *a display size*) here. The situation in the annotated sentence structure by NLP framework (in this case by spaCy NLP framework) is similar; however, it is not the same. There are 14 tokens, including the last token representing a dot. We can see that the verb (*to record*) – representing the predicate of the sentence – is the third token having the `VB` part-of-speech tag. There is a *nominal subject* dependency type (`nsubj`) connected to the mentioned verb – the subject of the sentence is represented by the first token (*we*), which is a pronoun as suggested by the `PRN` part-of-speech tag. However, direct object recognition needs to be custom handled. There is a *direct object* dependency type (`dobj`) referencing the sixth token (*number*), which is a noun as suggested by the `NN` part-of-speech tag. However, there is no second *direct object* dependency type. To find the second *direct object*, we need to navigate from the first *direct object* to the token connected via the *conjunct* dependency type (`conj`). Also, we need to expand other dependencies to get the full *direct object* text. Here, custom sentence patterns recognizing the sentence structure could be helpful.

In the next step of our algorithm, we use the method of grammatical inspection improved by our sentence patterns to process such annotated structures as the one in Fig. 3.1.

3.3 The Method of Grammatical Inspection

The method of grammatical inspection is based on the idea that the grammatical role of words (mapped to tokens) in a sentence of a requirement, i.e., *subject*, *object*, etc., indicates a representation of the requirement parts in a domain model. For example, both the subject and the object are candidates for an entity (a class – for our purpose, these two terms are interchangeable) or an attribute in a domain model.

On behalf of the method of grammatical inspection, we define sentence patterns following [99]. These patterns use the grammatical structure (primarily part-of-speech tags and dependencies). The whole text is checked against our collection of patterns, sentence by sentence. The extracted parts matching one of our patterns are stored in our internal model. Before we present the concrete patterns, we will first show the structure of our internal model to illustrate how we map the extracted parts of the text.

Although some approaches use machine learning techniques for the classification of requirements (e.g., functional vs. non-functional requirements) [130], for the purpose of model extraction and the subsequent semantic analysis, we use the approach of defining sentence patterns. One of the reasons is the lack of missing data for machine learning

[1] `https://spacy.io/models/en`

[2] Multiple direct objects are called *compound direct objects*.

model training because software requirements specifications are typically held as a private property of software companies.

3.4 Internal Model (Manager)

In the end, the information that we would like to extract from the requirements is a collection of entities (classes), their attributes, and relations between the entities. Thus, the goal is to transform the *syntactic analysis* performed on a text of the requirements into a *semantic form* – a domain model.

The output of our text mining process is an internal preliminary model represented by a quadruple $M = (E, G, R, S)$ where

- E is a set of elements,
- G is a set of element groups,
- R is a set of relationships, and
- S is a set of statements.

Each element $e \in E$ has the following properties:

root lemma the lemma form of the main word (token) representing element, e.g., *room* if we have element representing *hotel room*,

full lemma the lemma form of the whole element, e.g., *hotel room*,

candidate type indicates whether the element should be modeled as an entity or as an attribute or if it represents a value of some attribute,

is unique a flag indicating whether the requirements state that the element values are unique or not.

Each element group $g \in G$ clusters elements having the same *root lemma*; therefore, each element $e \in E$ is a part of exactly one element group g.

Each relationship $r \in R$ is a binary relation linking two elements $e_k, e_l \in E$. Typically, $k \neq l$; however, for recursive relationship, e_k could equals e_l. Each relationship $r \in R$ has the following properties:

source element occurrence representing $e_k \in E$; typically a subject,

target element occurrence representing $e_l \in E$; typically an object,

predicate the relationship name (represented by a verb),

is predicate negated a flag indicating if the predicate is negated or not,

type an optional value that indicates if a relationship represents a special relation, e.g., attribute ownership or generalization.

Element occurrence wraps an element and required properties for a relationship. The list of properties is as follows:

element a reference to element e,

multiplicity exactly one (`1`), zero or one (`0..1`), zero to many (`0..*`), one to many (`1..*`), or a custom range,

is multiplicity strict in some requirements, the multiplicity is explicitly mentioned; in some requirements, the multiplicity could be derived from the plural or singular nouns; in mentioned cases, this flag is set to `true`; in the rest cases, the default multiplicity zero to many is used, and this flag is set to `false`,

quantifier *each*, *none*, etc.

Finally, a statement $s \in S$ represents an expression about an element $e \in E$ when this expression is not representable by a relationship – there is not present the second element. In some cases, the missing second element could represent a *user* or a *system* in general. For example, *"A document can't be edited."* Each statement $s \in S$ has the following properties:

element representing $e \in E$,

quantifier *each*, *none*, etc.,

predicate the statement name (represented by a verb),

is predicate negated a flag indicating if the predicate is negated or not.

3.4.1 Internal Model (Manager) Constraints

The goal of the internal model (manager) is to store the extracted information and to provide constraints such as:

- text fragments representing the same semantic part of the model (e.g., *hotel room*) are stored as only one element e,
- text fragments listed in a configurable blacklist are not processed,

- elements marked as entity candidates (i.e., because we already found this semantic indicator in the previous text parts) can not be changed to attribute candidates; the opposite change from attribute candidate to entity candidate is fully legit because we can find that original attribute candidate has custom attributes or other entity symptoms in the different parts of the text.

Moreover, for selected problems detection, we use custom clustering based on selected parts of the model M. Details are provided in the corresponding chapters where such an approach is used.

The blacklist of text fragments excludes such parts from storing them in the model. The typical reason is that the meaning of these text fragments is too general. For example, we do not want some introductory sentence like *"This document describes requirements of..."* to generate entities *document* and *requirement*, and connect them via the relationship *describe*. On the other hand, in the requirements describing a document management system, the word *document* will represent one of the key entities. Because such exclusion is highly context-dependent, we leave the blacklist configurable. The default blacklist has been created manually by our previous experience with requirements processing.

3.5 Suitable Patterns

Now, we know the target structure that is the output of the grammatical inspection method, so we can present the concrete suitable patterns that we use to create and populate the internal model.

We divide this section into sub-sections representing specific text structure recognition. It is not our goal to present all the patterns we use here; we present illustrative examples.

3.5.1 Triplet Recognition

The basic semantic information of sentences is hidden in triplets. In our case, a triplet consists of a subject, a predicate, and an object, similar to the *Resource Description Framework* (RDF). Such triplet is represented by a *relationship* with a *source element* and a *target element* in our defined model M.

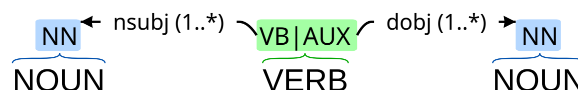


Figure 3.2: Basic triplet pattern.

In Fig. 3.2, there is an example structure of the annotated text segment that corresponds to a triplet recognition pattern. The predicate is represented by a verb (the part-of-speech tag property of a token is `VB`) or by an auxiliary (for example, when we consider the verb *to be*; the part-of-speech tag property of a token is `AUX`). Next, there are two already introduced dependency types. The first one is annotated as `nsubj`, and the target token

represents the *subject* in our triplet. The second one is annotated as `dobj`, and the target token represents the *object* in our triplet.
When such a pattern is matched, we can identify:

- entity candidate element e_1 represented by the nominal subject part of a sentence,
- entity candidate element e_2 represented by the dependency object part of a sentence, and
- relationship r where the predicate (name) of the relation is represented by the lemma of the verb token.

We can notice that the dependency type label includes the `1..*` part. It represents a multiplicity of the dependency type. We require at least one subject and one object. In any case, there can be more subjects or objects. In such cases, we generate more triplets with the same predicate.

The second level of multiplicity is directly represented by the concrete subject or object. We need to distinguish between sentences *"the business group owns a hotel"* and *"the business group owns hotels."* This information is a part of the part-of-speech tag where `NN` represents a singular noun and `NNS` represents a plural noun. Anyway, this is not enough, and we need to consider other linguistic expressions of plurality, e.g., *"the business group could own more than one hotel"* or *"the business group owns at least one hotel"*.

Because this pattern is too general, it is wise to use it as the last one when other patterns considering more specific structures are not matched.

3.5.2 Triplet Recognition – Challenges

The pattern presented in Fig. 3.2 reflects the basic sentence structure. However, usually, the situation is not that simple. We need to face the challenges in the form of passive voice, modificators in the form of prepositions, indirect subjects, etc.

In Fig. 3.3, there is an example of a passive voice sentence. We are now curious about the `nsubjpass` dependency type instead of the `nsubj` dependency type used in the active voice. We can note that we also miss the direct `dobj` dependency type. This time, there is the `pobj` dependency type that presents an object of a preposition. Also, our object (the word *guest*) is not directly connected to the predicate (the verb *place* (the lemma form)), so we need to check another dependency type called `agent`.

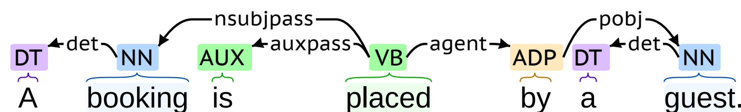


Figure 3.3: Example of the passive voice #1.

When we match this pattern, we consider the information as a triplet transformed into an active voice. In the case of the example in Fig. 3.3, the result triplet is ⟨*guest*, *place*, *booking*⟩.

A similar situation occurs when the `agent` dependency type is replaced by the `prep` dependency type. The example is shown in Fig. 3.4. This time, we expect the triplet ⟨*room*, *relate (to)*, *booking*⟩.

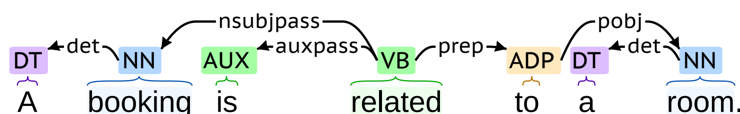


Figure 3.4: Example of the passive voice #2.

To make it easier, in the examples of triplets recognition, we present the root token (word) representing entity or attribute candidates. However, as defined in our internal model structure in Section 3.4, we also recognize the full lemma. The entity or attribute name is often a composition of multiple nouns, a noun and an adjective, or both. This situation can be recognized by patterns checking the `compound` dependency type and the `amod` (*adjectival modifier*) dependency type. Both representatives are present in Fig. 3.1: the *serial number* (adjective + noun) and the *display size* (compound nouns).

3.5.3 Attributes Recognition

In this category, we present three illustrative representatives.

3.5.3.1 Attributes Recognition Pattern – To Have

In Fig. 3.5, there is an attribute(s) mapping using the verb *have*. In this case, the root triplet of the sentence is ⟨*user*, *has*, *username*⟩. The subject (*user*) and the object (*username*) are mapped to elements in our internal model. The *user* is mapped as an entity candidate e_1 and the *username* is mapped as an attribute candidate e_2.

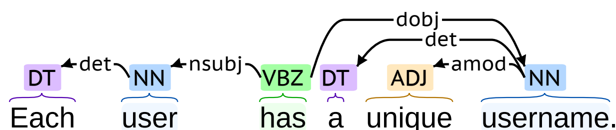


Figure 3.5: Example of a sentence with attribute candidate and the verb *have*.

Furthermore, we can see that the *username* token targets the *unique* token via the `amod` (*adjectival modifier*) dependency type. Therefore, we can set the *is unique* property of e_2 to `true`.

Next, we can see the *user* token targets the *each* token via the `det` (*determiner*) dependency type. Therefore, we can set the *quantifier* property of the element occurrence representing the e_1 in a corresponding relationship. A note – in our model, we still map the

relationships between an entity candidate and the potential attribute candidates because any attribute candidate could be changed into an entity candidate during further analysis of the text (as discussed in Section 3.4.1). The second reason is that we can benefit from the mapped properties of the relationship. When the model is serialized, for example, into a UML class diagram, we use the standard notation of a class and attributes and the relationship is not present in the serialized diagram.

You can see that the sentence structure also conforms to the previously shown pattern regarding the general triplet. So, we need to check the lemma of the verb first to prioritize the attribute pattern over the general triplet pattern.

3.5.3.2 Attributes Recognition Pattern – Of

Let us recall the example of the annotated sentence from Section 3.2 in Fig. 3.6 as another example of the attributes recognition pattern. This time, the attribute indication hint comes from the preposition *of* (the part-of-speech tag property of the corresponding token is `ADP` – it means *adposition*[3]). Following the already presented triplet recognition pattern, we found a subject (the pronoun *we*), predicate (the verb *record*) and two objects (in their full form – *number* and *display size* – using the `compound` dependency type). The second object has an outgoing dependency with the `prep` (*prepositional modifier*) dependency type targeting the preposition *of*. Following this preposition, we can find the *television* token using the `pobj` dependency type that represents an object of a preposition.

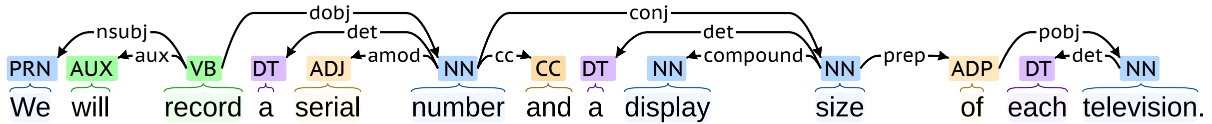


Figure 3.6: Example of a sentence with attribute candidates.

To summarize the pattern matching – the object of preposition (the *television* token) is an entity candidate element e_1, and the original triplet objects (*number* and *display size*) are attribute candidate elements e_2 and e_3. Because e_2 and e_3 are considered attributes, we create two relationships:

- r_1 connecting e_1 and e_2 with the substituted predicate *to have* and
- r_2 connecting e_1 and e_3 with the substituted predicate *to have*.

We do not consider the combination of the original triplet subject (the personal pronoun *we*) and predicate (the verb *to record*) semantically rich; therefore, we do not include them in the model. Such combinations are part of the blacklist presented in Section 3.4.1. In addition to the verb *to record*, we can also consider the verbs *to register*, *to keep*, or *to write down*.

[3] Adposition is a cover term for prepositions and postpositions.

3.5.3.3 Attributes Recognition Pattern – Indirect Subject

As mentioned in the triplet recognition challenges, we need to take into account indirect subjects, too. Such a case is shown in Fig. 3.7. In this example sentence, we once again skip the combination of the original triplet subject (the personal pronoun *we*) and predicate (the verb *to keep*) because it is not semantically rich for us as discussed in the previous paragraph.

We would like to extract triplets in the form ⟨*booking*, *have*, *start date*⟩ – with the substituted predicate *to have* – and variants for other attributes (*end date* and *price*). This sentence structure can be matched by a pattern detecting the `prep` (preposition) dependency type of the original triplet predicate. When the preposition is matched, the second step is to match the expected entity candidate (in our case – *booking*) using the `pobj` (object of a preposition) dependency type.

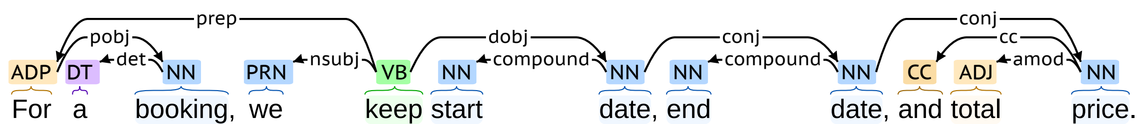


Figure 3.7: Example of a sentence with an indirect subject.

3.5.4 Hierarchy Recognition

In the UML class diagram, we can model the generalization relationship representing a concept of inheritance as recalled in Section 2.3.1.2. In further model processing, we can benefit from mapping entity hierarchy using the recognition of inheritance in the text.

In Fig. 3.8, there is an example sentence representing the hierarchy of entities. The hierarchy recognition pattern comes from the verb that has to be a form of the verb *to be*. Next, the pattern checks the `nsubj` (nominal subject) dependency type targeting element e_1, and there has to be the `attr` (attribute) dependency type instead of the `dobj` (direct object) dependency type targeting element e_2. Both e_1 and e_2 are mapped as entity candidate elements – e_1 represents the base entity and e_2 represent the sub-entity. There should be more than one sub-entity. We can find the other candidates via the `conj` (conjunct) dependency type, as already illustrated.

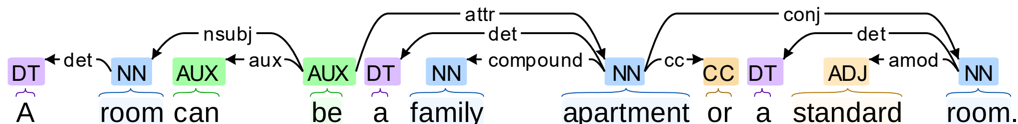


Figure 3.8: Example of a sentence representing hierarchy of entities.

In the words of triplets, we can identify two triplets here: ⟨*family apartment*, *is*, *room*⟩ and ⟨*standard room*, *is*, *room*⟩. When processing or serializing the internal model, such relationships could be interpreted as a concept of inheritance.

3.5.4.1 Hierarchy Recognition – Challenges

The verb *to be* is so common; therefore, we introduce a check of element type candidates. Let us consider a similar example sentence (in Fig. 3.9) to the one discussed in the previous paragraphs – *"A status of each user can be online, offline, or away."* This time, the `nsubj` dependency type targets the *status* token that is (because of the *attribute recognition pattern* with the preposition *of*) an attribute element candidate e_1. Thus, the *online*, *offline*, and *away* tokens are recognized as value-type elements connected with e_1.

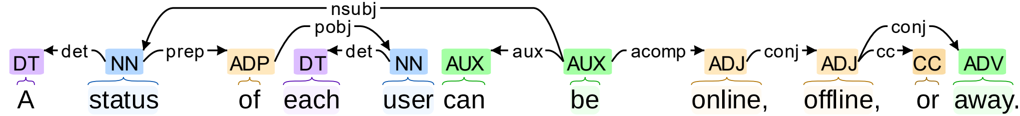


Figure 3.9: Example of a sentence representing attribute values recognition.

3.6 Evaluation

In the following chapters, we discuss our approach to each problem defined in the goals of this thesis. To evaluate our proposed methods, we define a data set in the next section. Moreover, as proposed in Section 1.4.1 (Research Approach), we iteratively implement selected methods to extend our created software artifact – system TEMOS. Therefore, we introduce the general pipeline of our system TEMOS in this section, too.

3.6.1 Data Set

In this section, we describe the structure of the used data set. This same data set is then used across all chapters where we evaluate the methods over a collection of different documents. We have collected textual requirements from four different sources:

- a set of requirements from the PUblic REquirements (PURE) dataset [39] (prefixed with uppercase P in the evaluation tables),
- a set of requirements collected by Hayes et al. [56] (prefixed with uppercase H in the evaluation tables),
- a set of requirements provided by [24] (prefixed with lowercase g in the evaluation tables),
- a custom collection of requirements found in the Internet (prefixed with uppercase C in the evaluation tables).

From the existing data sets, we select such requirements that satisfy a domain description (for example, we have excluded the specifications describing the physical signals). We preserve the original file of the requirements (`TXT`, `PDF`, `RTF`, `DOC`, or `HTML`) and

add a plain text conversion. We have converted other formats into `TXT` text files and improved the converted text – fixing UTF-8 formatting, removing page headers and footers, fixing broken list numbering (e.g., due to interruption by an image or another page in the original document), and fixing whitespace formatting (e.g., unnecessary double spaces or whitespace characters at the end of the line). Moreover, we have converted selected original files to the Markdown syntax (including a manual fix of headers and lists), too.

Additionally, for some of the requirements files where it was recognizable, we have extracted part of functional requirements or functional specification relevant parts. We have published this data set using the Zenodo open repository – it is available via DOI identifier (`10.5281/zenodo.7897601`) [A.11].

For an idea of the complexity of the texts, we provide basic statistics. In Table 3.1, we describe each input data text with the number of sentences, the number of words, the average number of words per sentence, and two indices of readability – the *automated readability index* (ARI) and the *Gunning fog index* (GFI) – that we obtained using the *Free Text Complexity Analyzer* online tool[4]. One can find details about readability indices, e.g., in [126].

[4]`https://www.lumoslearning.com/llwp/free-text-complexity-analysis.html`

Table 3.1: Data set statistics.

Case	**Words**	**Sentences**	**WpS**	**ARI**	**GFI**
g02-federalspending	2,089	98	21.32	8.4	12.3
g03-loudoun	1,579	57	27.70	14.2	15.0
g04-recycling	1,287	51	25.24	10.1	14.2
g05-openspending	1,640	53	30.94	12.6	11.0
g08-frictionless	1,746	66	26.45	11.8	14.8
g10-scrumalliance	2,571	99	25.97	9.9	9.2
g11-nsf	1,749	73	23.96	9.0	13.6
g12-camperplus	1,397	53	26.36	10.5	9.7
g13-planningpoker	1,457	53	27.49	10.4	10.6
g14-datahub	1,841	67	27.48	11.0	10.0
g16-mis	1,536	67	22.93	12.1	12.9
g17-cask	1,627	64	25.42	11.4	16.8
g18-neurohub	2,200	101	21.78	9.4	13.5
g19-alfred	2,441	138	17.69	7.2	10.3
g21-badcamp	1,889	70	26.99	12.1	15.3
g22-rdadmp	2,246	83	27.06	12.4	16.5
g23-archivesspace	875	56	15.63	7.6	10.2
g24-unibath	1,464	52	28.15	13.8	17.8
g25-duraspace	2,015	100	20.15	9.5	16.2
g26-racdam	2,122	100	21.22	9.9	15.1
g27-culrepo	3,316	120	27.63	13.8	17.3
g28-zooniverse	1,060	60	17.67	9.2	12.6
P01. Blit	535	48	11.15	5.0	7.9
P02. CS179G – ABC Paint Project	1,199	66	18.17	8.0	10.4
P03. eProcurement	1,683	90	18.70	11.4	12.3
P04. Grid 3D	196	11	17.82	4.9	7.0
P05. Home 1.3	1,121	87	12.89	6.4	10.0
P06. Integrated Library System	1,974	79	27.80	8.3	9.2

Case	Words	Sentences	WpS	ARI	GFI
P07. Inventory	4,657	500	9.31	3.5	8.5
P08. KeePass Password Safe	466	36	12.94	4.2	7.9
P09. Mashbot	619	26	23.81	8.8	11.6
P10. MultiMahjong	1,759	88	19.99	9.3	9.0
P11. Nenios	944	82	11.51	6.6	8.3
P12. Pontis 5.0 Bridge Management System	4,395	221	19.89	11.5	13.5
P13. Public Health Information Network	2,988	110	27.16	17.4	16.9
P14. Publications Management System	2,546	61	41.74	9.1	6.9
P15. Puget Sound Enhancements	2,046	93	22.00	7.0	9.0
P16. Tactical Control System	5,855	296	19.78	7.5	10.9
P17. Tarrant County Integrated Justice Information System	1,763	134	13.16	13.1	11.7
P18. X-38 Fault Tolerant System Services	5,455	358	15.24	12.5	14.7
H01. CCHIT	2,430	112	21.70	12.7	13.8
H02. CM1	514	30	17.13	10.2	12.7
H03. InfusionPump	2,956	249	11.87	10.4	12.2
H04. Waterloo	11,810	664	17.79	9.0	10.8
C01. Amazing Lunch Indicator	3,241	93	34.85	5.4	8.7
C02. EU Rent	492	43	11.44	4.1	7.8
C03. FDP Expanded Clearinghouse Pilot	464	39	11.90	9.9	10.5
C04. Library System	1,779	128	13.90	6.1	10.7
C05. Nodes Portal Toolkit	2,239	153	14.63	7.8	8.8
C06. Online National Election Voting	3,240	264	12.27	6.1	5.6
C07. Restaurant Menu & Ordering System	896	46	19.48	7.9	11.6

Legend: **WpS** – average number of words per sentence, **ARI** – Automated Readability Index, **GFI** – Gunning Fog Index

3.6.2 Textual Modeling System (TEMOS)

In the schema in Fig. 3.10, we present the general pipeline of our approach according to the defined Algorithm 3.1. This time, we also show the integration of our tool TEMOS. We introduce details about TEMOS in Chapter 9 (Created Artifact and Models Generation). However, as proposed in Section 1.4.1 (Research Approach), we iteratively implement selected methods to extend TEMOS; therefore, we briefly describe the pipeline here so we can refer to it.

First, we use the NLP framework pipeline as described in Section 3.2. The created annotated text is then processed using our defined patterns (Section 3.5) – displayed as the *core analyzer* in the scheme. Based on that, the internal model is created (Section 3.4). Next, each researched problem is supported by a custom analyzer. This is the part in the bottom right corner of the schema that is iteratively extended in the following chapters. Finally, analyzers and the created internal model produce models to export (e.g., UML class diagram) and generate warnings and questions regarding text issues.

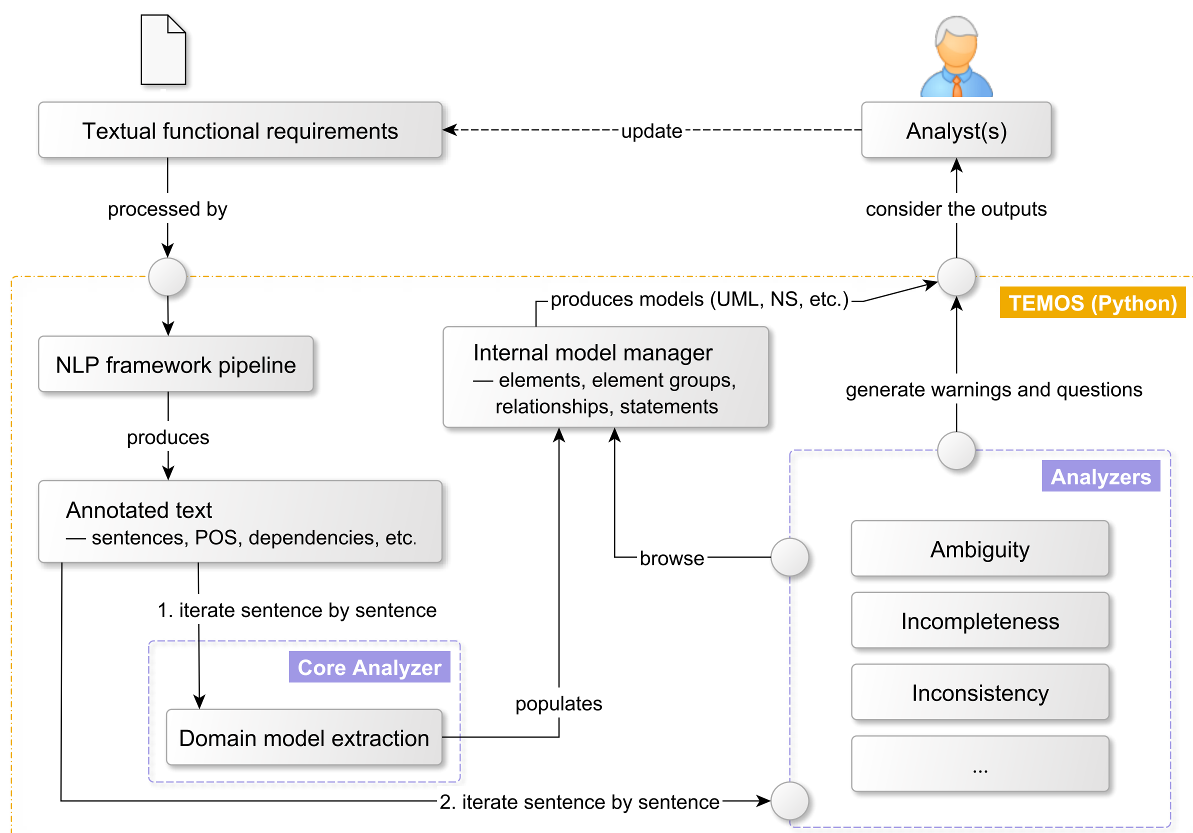


Figure 3.10: TEMOS pipeline.

Chapter 4

Problem of Ambiguity in Textual Requirements Specification

This chapter reflects our publications:

- Šenkýř, D.; Kroha, P. Patterns of Ambiguity in Textual Requirements Specification. In: *New Knowledge in Information Systems and Technologies*. Springer, Cham, 2019. [A.9]
- Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies*. SciTePress, Madeira, 2018. [A.10]

In this chapter, we investigate the ambiguity problem in textual requirements specifications. We applied the structural ambiguity approach and extracted some patterns to indicate this kind of ambiguity. We show that the standard linguistics methods are not enough in some cases, and we describe a class of ambiguity caused by coreference that needs an underlying domain model or a knowledge base to be solved. Part of our implemented solution includes working with facts and rules acquired from OCL conditions of the domain model.

4.1 Motivation

Textual formulated requirements specifications are necessary as a base of communication between the customer, the user, the domain expert, and the analyst. Unfortunately, any text suffers from ambiguity, incompleteness, and inconsistency.

In this chapter, we focus on the problem of ambiguity that is caused by the phenomenon that natural language is inherently ambiguous, i.e., one word can have multiple meanings according to the context it is used. Requirements specifications have to be unambiguous because, in general, any ambiguous requirement is not verifiable, and there is a risk that programmers implement the meaning that was not intended by the specification. According to IEEE standard [67], an SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation.

Information retrieval operates with a process called *word sense disambiguation (WSD).* It has to determine the correct sense of the given word in the context it was used in the text. Usually, this problem is solved by co-occurrence, i.e., the context is specified by words occurring in the same text in some connection to the word whose ambiguity has to be decided. The goal is to identify the unique meaning of each word.

Text mining operates with a more complex version of the ambiguity problem in text documents that can be called *sentence sense disambiguation.* The goal is to identify the unique meaning of each sentence.

In our tool TEMOS, we construct and maintain a glossary that defines and explains the meaning of concepts (selected words) used in a textual requirements specification. The word meaning is derived from the online dictionaries, and the glossary is discussed by the customer, the user, the domain expert, and the analyst during the analysis of requirements. Using a glossary of terms, we can check whether different terms are mistakenly used for the same concept by checking synonyms in online semantic networks.

In this chapter, we explain why it is not enough for our purpose and how we solved it. In common, the ambiguity of sentences is given by structural ambiguity that was investigated by linguists. We have constructed the corresponding patterns and described them in Section 4.3.

Practically, we found that in most cases, ambiguity is combined with the coreference problem in textual requirement specifications. Coreference resolution has to find all expressions that refer to the same entity in a text. In textual requirements specification, it often happens that a pronoun in a sentence refers to a specific noun from the previous

clause. If there is more than one noun as a possible candidate, it is to decide which is the right one.

We propose a new method to solve the problem mentioned above because we have found that the text layer used in computational linguistics is not strong enough for our purpose. The roots of ambiguity in our investigated textual requirements specifications concern not only the linguistic part of requirements but also the semantics of the underlying model, as we show in Section 4.6.

4.2 Problem Statement and Related Work

The common ambiguity issues in various texts are discussed by linguists [79], [58], [2], [17].

We target the issue of ambiguity in textual requirements specification from two perspectives – the *ambiguity of words* (also called the *lexical ambiguity*) and the *ambiguity of sentences* (also called the *structural ambiguity* or the *syntactic ambiguity*). We explain the difference in the following subsections.

4.2.1 Ambiguity of Words

Natural languages are inherently ambiguous. Ambiguity arises whenever a word or an expression can be interpreted in more than one way. If the expression is a single word, we speak about word disambiguation to resolve it. The reason is that natural languages use:

- *homonyms*, i.e., words that are spelled alike but have different meanings,
- *synonyms*, i.e., different words that have the same meaning.

The user, the customer, and the domain expert understand the domain and know that two or more different words have the same meaning (synonyms) or what homonym meaning is the right one in the given context. However, the analyst does not know that in some cases and can model them as unique entities. It happens that project teams are places in different countries where different terminology has been used.

This word disambiguation we solved with the help of a *glossary* similar to the proposal in [3]. Moreover, we support the glossary by thesaurus that is freely available on the Internet. As a result, questions are generated by asking stakeholders whether they agree with the equivalence found. Finally, the result is stored in the glossary again.

Wang et al. [128] presented a different approach via a *ranking method* to deal with the problem of *overloaded* (it refers to different semantic meanings) and *synonymous concepts* (several different concepts are used interchangeably to refer to the same semantic meaning).

4.2.2 Ambiguity of Sentences

The ambiguity of sentences is a more complex problem because the parsing structure of a sentence is not unique in all cases, i.e., identical sentence text segments (composed from

the same words – even if these words would have unique meaning), can have a different meaning that depends on their position in the parsing tree of the sentence.

This problem has been discussed in detail by computational linguists as the problem of structural ambiguity, e.g., in [89]. The complete analysis is given, e.g., in [89] and [57], and it is out of the scope of this chapter. Our contribution is that we have developed a set of patterns that indicate possible ambiguities in textual requirements and, in some cases, offer possible solutions. Actually, the semantics of the world is very complex, so we do not think it is possible to solve the ambiguity problem automatically in all cases.

Some of the ambiguities can be indicated by tools of computational linguistics, as shown in Section 4.3. Some others can not, and they are shown in Section 4.6.

4.3 Our Approach – Patterns of Structural Ambiguity

In this contribution, we focus on the ambiguity of sentences that should be solved by *patterns*. The candidates for structural ambiguity patterns can be derived from the following linguistic features: *attachment ambiguity* and *analytical ambiguity* [57].

Legend. Within the following examples of patterns, we reuse the presented approach of grammatical inspection using *part-of-speech* and *dependency* tags as presented in Chapter 3. In some cases, it is possible to have multiple part-of-speech tag candidates. We use the pipe symbol (`|`) in these situations. This symbol means that a pattern is matched if and only if exactly one part-of-speech tag from the defined candidates is present in the grammatical sentence structure.

Often, the subjects and the objects are represented by a composition of several nouns. Therefore, we introduce a shortened notation `NN*`, which means that at least one noun is required, and there should also be more nouns composited via the *compound* relation type.

In the following examples, we present a generated warning structure when the concrete pattern is matched – it means that the sentence structure is suspicious from our point of view. In the warning structure, we refer to the parts of the pattern. The referred part is then replaced by an actual word from the investigated sentence.

4.3.1 Patterns of Attachment Ambiguity

Attachment ambiguity represents sentences with more than one node to which a particular syntactic constituent can be attached [57]. The nature of the constituent is clear, but it is not clear where to put it. Below, we attach selected examples with sample patterns that help to reveal such ambiguity.

4.3.1.1 Pattern #1 (Prepositional Phrase Modifier)

A prepositional phrase may either modify a verb or an immediately preceding noun phrase, e.g., *"He saw the man with field glass."* The interpretation of the sentence can vary – *"he*

saw the man using his field glass" or *"he saw the man who has a field glass"*. The pattern matching such sentence structure is in the following Fig. 4.1.

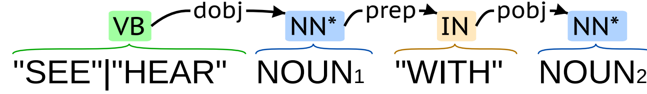


Figure 4.1: Pattern #1 (prepositional phrase modifier).

- *Example of generated warning*: Ambiguity – "`VERB` `NOUN`$_1$ via/using `NOUN`$_2$" vs. "`VERB` `NOUN`$_1$ that has/have `NOUN`$_2$".

4.3.1.2 Pattern #2 (Preposition Phrase)

A prepositional phrase may have more than one noun phrase available to attach it to [57], e.g., *"The door near to stairs with the 'Members Only' sign..."* The interpretation can vary – *"door with the 'Members Only' sign"* or *"stairs with the 'Members Only' sign"*? The pattern matching such sentence structure is in the following Fig. 4.2.

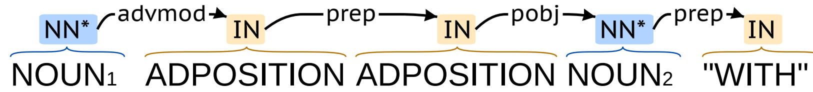


Figure 4.2: Pattern #2 (preposition phrase).

- *Example of generated warning*: **"with"** – ambiguity between `NOUN`$_1$ and `NOUN`$_2$.

4.3.1.3 Pattern #3 (Relative Clause)

A similar problem to the previous pattern can also occur with relative clauses. Relative clauses may have a relation to more than one noun of the main clause [57], e.g., *"The door near to stairs that had the 'Members Only' sign..."*. The pattern matching such sentence structure is in the following Fig. 4.3.

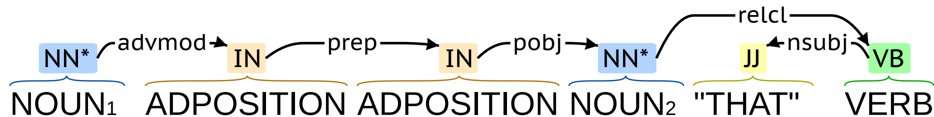


Figure 4.3: Pattern #3 (relative clause).

- *Example of generated warning*: **"that"** – ambiguity between `NOUN`$_1$ and `NOUN`$_2$.

4.3.1.4 Pattern #4 (Subsentence Attachment)

When a sentence contains a subsentence, both may include places for the attachment of a prepositional phrase or adverb [57], e.g., *"He said that she had done it yesterday."* The interpretation can vary – *"he said it yesterday"* or *"she had done it yesterday"*? The pattern matching such sentence structure is in the following Fig. 4.4.

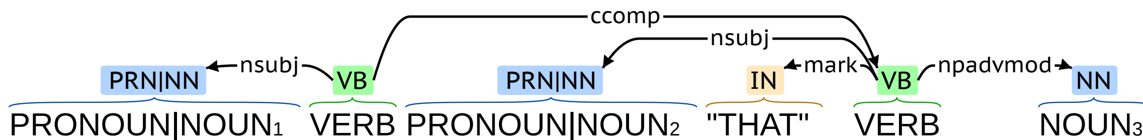


Figure 4.4: Pattern #4 (subsentence attachment).

- *Example of generated warning*: `NOUN`$_3$ – ambiguity between "`PRONOUN|NOUN`$_1$ ..." and "`PRONOUN|NOUN`$_2$ ..."

4.3.1.5 Pattern #5 (Adverbial Position (1))

An adverbial placed between two clauses can be attached to the verb of either [57], e.g., *"The lady you met now and then came to visit us."* The pattern matching such sentence structure is in the following Fig. 4.5.

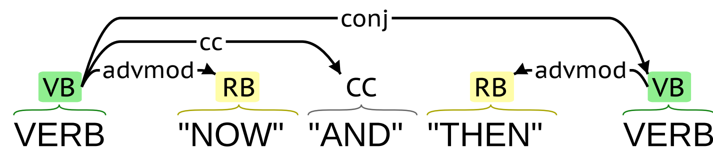


Figure 4.5: Pattern #5 (adverbial position (1)).

- *Example of generated warning*: A possible **"now and then"** ambiguity – do you mean **"sometimes"** or not?

4.3.1.6 Pattern #6 (Adverbial Position (2))

"A secretary can type quickly written reports" is an example of adverb placement ambiguity [57]. The pattern of this example is a little bit complicated because of a bad annotated word *written* as a verb instead of an adjective, as shown in the following parsing result in Fig. 4.6.

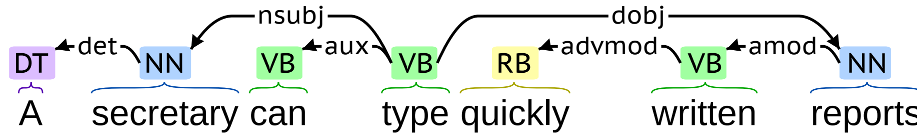


Figure 4.6: Pattern #6 (adverbial position (2)) – example sentence.

However, when we replace the word *written* with *handwritten*, then the word *handwritten* is correctly recognized as an adjective. Based on this observation, we have created the corresponding pattern presented in Fig. 4.7.

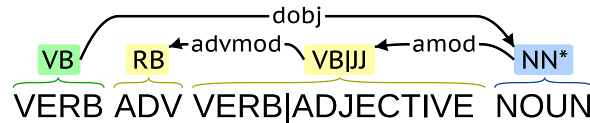


Figure 4.7: Pattern #6 (adverbial position (2)).

- *Example of generated warning*: Please consider a possible ambiguity of the adverb `ADV`.

4.3.2 Patterns of Analytical Ambiguity

Analytical ambiguity represents sentences with more than one possible syntactic analysis, and the nature of the constituent is itself in doubt [57]. It is also presented in the implementation where the misrecognition of a part-of-speech tag is itself a sign of ambiguity because the NLP pipeline is unsure of the choice.

4.3.2.1 Patterns #7 and #8 (Reduced Restrictive Relative Clause (1) and (2))

Reducing a pronoun of the restrictive relative clause of the example sentence *"I want the box on the table"* can have the meaning *"I want the box that is on the table"* or *"I want the box to be on the table"* [57]. The pattern matching such sentence structure is in the following Fig. 4.8.

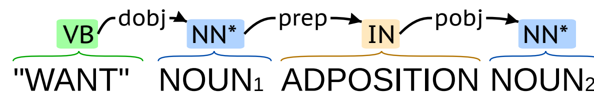


Figure 4.8: Pattern #7 (reduced restrictive relative clause (1)).

"I want the chair next to the bed" is an example of a variation with the *adverb*. The pattern matching such sentence structure is in the following Fig. 4.9.

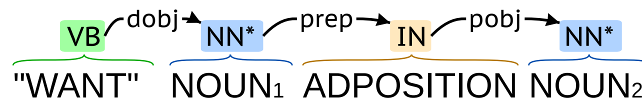


Figure 4.9: Pattern #8 (reduced restrictive relative clause (2)).

- *Example of generated warning*: `NOUN`$_1$ – that is/are "`ADPOSITION` ..." or to be "`ADPOSITION` ..."?

4.3.2.2 Pattern #9 (Present Participle vs. Adjective)

Distinguishing a present participle from an adjective is another source of ambiguity [57], e.g., *"We are writing a letter."* and *"Pen and pencils are writing implements."* The pattern matching such sentence structure is in the following Fig. 4.10.

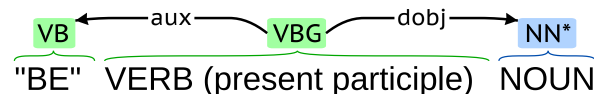


Figure 4.10: Pattern #9 (present participle vs. adjective).

- *Example of generated warning*: Please consider a possible ambiguity of the word `VERB`.

4.3.2.3 Pattern #10 (Participle)

The participle could be attached to the object either as a reduced restrictive relative clause or as a verb complement, e.g., *"The manager approached the boy smoking a cigar."* or *"The manager caught the boy smoking a cigar."* The pattern matching such sentence structure is in the following Fig. 4.11.

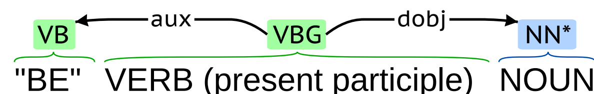


Figure 4.11: Pattern #10 (participle).

- *Example of generated warning*: Please consider a possible ambiguity of the word `VERB`$_2$.

4.4 Our Approach – Glossary Construction and Synonyms Resolving

The key part of the textual requirements specification analysis is building a *glossary of terms*. A glossary should describe all the domain-specific terms used in the requirements. Every class or attribute candidate (as proposed in Section 3.4) is automatically introduced as a term of the glossary. This is the default basic generated version of the glossary that should be extended by stakeholders – terms can be added or corrected.

Concerning the glossary and the text disambiguation process, we can support stakeholders on two levels.

First, we can use online dictionaries to get a default meaning (definition) of terms that can then be corrected by stakeholders if needed. We use the Babel [86] and Wordnik online dictionaries for that. Authors claim that Wordnik is the world's biggest online English dictionary by number of words [129].

Second, handle synonyms of terms. Because requirements specifications could be constructed and updated by distributed teams, multiple terms for the same concept could arise in the text.

We can support stakeholders here using the online thesaurus service. Even the already mentioned Wordnik provides a collection of synonyms for the queried term. In our implementation, we use ConceptNet [113] – the online semantic network – where we can look for synonyms in a collection of terms of our glossary. ConceptNet API also provides a list of synonyms of a particular term, such as Wordnik. Moreover, ConceptNet API provides a relation called *relatedness of a particular pair of terms* [112] that returns a weighted value on the scale $[0; 1]$. In our implementation, a user can set a threshold value for the accepting terms as synonyms.

Listing 4.1: ConceptNet – relatedness of a particular pair of terms – call the endpoint.

```
GET
https://api.conceptnet.io/relatedness
   ?node1=/c/en/booking
   &node2=/c/en/reservation
```

Let us present an example of a hotel management system where two terms representing the same domain concept have been introduced – a *booking* and a *reservation*. Calling the *relatedness of a particular pair of terms* endpoint, as shown in Listing 4.1, results in the response presented in Listing 4.2 saying that these two terms are *72 percent* related.

Listing 4.2: ConceptNet – relatedness of a particular pair of terms – response.

```
{
    "@context": [
        "http://api.conceptnet.io/ld/conceptnet5.7/context.ld.json"
    ],
    "@id": "/relatedness?node1=/c/en/booking&node2=/c/en/reservation",
    "value": 0.721,
    "version": "5.8.1"
}
```

As a result of the presented synonyms check, a user can update the text based on the synonyms suggestion or at least group the synonyms as one term in the process of model creation.

4.5 Experiments and Results

Although we have presented examples investigated by linguists [57][89] (taken from novels, newspapers, and jokes), the created patterns are easily applicable to the text of requirement, too. For example, the pattern Nr. 2 (*preposition phrase*) could catch sentences like *"The button next to the warning box with the red border..."* that could imply multiple meanings. Does the button have the red border? Or does the warning box have the red border?

In the development of our other project, our colleague wrote this sentence as a definition of done of one request ticket: *"Having a checkbox next to the save and cancel buttons that says something like 'create another'."* We have tested this sentence in Grammarly [68], the writing assistant. The result is shown in Fig. 4.12. As you can see, Grammarly is confused about the subject. Such sentence structure is matched by our pattern Nr. 3 (*relative clause*), and the warning is generated. We motivate our users to use simple sentences where the subject is directly mentioned without a reference.

Having a checkbox next to the save and cancel buttons that says something like 'create another'.

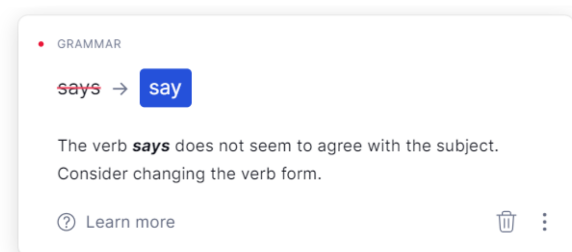


Figure 4.12: Ambiguous parsing structure in Grammarly.

Table 4.1 summarizes the results of experiments of our presented ambiguity detection approach using the data set introduced in Section 3.6. The results include the following:

- the frequency of matched patterns of structural ambiguity defined in this chapter,
  - we include only the patterns that have been matched,
  - the table uses this legend:

    PP pattern #10 (participle),
    PPA pattern #9 (present participle vs. adjective pattern),
    RRRC patterns #7 and #8 (reduced restrictive relative clause (1) and (2)),
    RC pattern #3 (relative clause),
    PPH pattern #2 (prepositional phrase),
    AP pattern #6 (adverbial position pattern (#2)),
- the frequency of synonyms hints:
  - some of them can be interpreted as true positives, e.g., the case called *g12-camperplus* (specifying camp organization information system) generated synonyms hint consider entities `child` and `kid` that seems to be used interchangeably in the specification; however, for most of the specifications, a domain expert should judge, so we leave the hints without classification,
  - for comparison with the number of generated synonym hints, one can find numbers of generated entities and attributes for cases from the used data set in Chapter 8.

We interpret the results in the following way.

Patterns When some of our presented pattern is matched, it means that the formulation could be ambiguous. We recommend to rewrite the sentence following the provided hint.

Synonyms We encourage users of our system to create a glossary. Among terms clarifying, a glossary is useful to process generated synonyms hints. A user of our system can interpret them as follows:

- if the two terms of generated hint represent synonyms, we recommend selecting one and using it strictly across the whole document, documenting it in a glossary, and mentioning the synonym variant in a glossary to make it clear for other users,
- if the generated hint is recognized as a false positive, we recommend making sure that both terms are part of a glossary and they are documented.

Table 4.1: Evaluation of methods detecting ambiguity.

Case	PP	PPA	RRRC	RC	PPH	AP	Synonyms
g02-federalspending	0	4	7	0	0	1	2
g03-loudoun	0	2	0	1	0	1	1
g04-recycling	1	2	1	2	1	0	0
g05-openspending	0	3	0	1	0	1	0
g08-frictionless	5	2	3	0	0	1	2
g10-scrumalliance	1	1	3	0	0	3	1
g11-nsf	0	3	2	0	2	1	2
g12-camperplus	0	5	0	0	1	1	1
g13-planningpoker	0	6	1	0	0	0	1
g14-datahub	1	4	0	0	0	2	0
g16-mis	0	2	1	0	1	0	1
g17-cask	1	0	0	1	0	1	0
g18-neurohub	10	2	0	0	0	4	4
g19-alfred	3	4	1	0	2	2	2
g21-badcamp	0	6	0	1	1	1	2
g22-rdadmp	0	0	0	0	1	1	3
g23-archivesspace	1	1	0	0	0	0	0
g24-unibath	0	1	1	0	0	1	0
g25-duraspace	1	0	1	2	0	0	3
g26-racdam	0	2	4	0	0	0	0
g27-culrepo	4	3	2	4	1	1	5
g28-zooniverse	4	0	1	0	0	0	0
P01. Blit	1	0	0	0	1	0	1
P02. CS179G – ABC Paint Project	0	0	0	0	2	1	9
P03. eProcurement	1	4	0	1	2	0	13
P04. Grid 3D	0	1	0	0	0	0	1
P05. Home 1.3	1	0	0	0	0	0	1
P06. Integrated Library System	1	1	0	0	2	0	21

Case	PP	PPA	RRRC	RC	PPH	AP	Synonyms
P07. Inventory	2	1	0	0	3	1	4
P08. KeePass Password Safe	0	0	0	0	0	0	5
P09. Mashbot	0	0	0	0	1	0	1
P10. MultiMahjong	2	2	0	1	3	1	8
P11. Nenios	0	0	0	1	1	0	4
P12. Pontis 5.0 Bridge Management System	10	0	0	0	1	0	23
P13. Public Health Information Network	0	0	0	0	1	2	31
P14. Publications Management System	1	0	0	0	0	1	4
P15. Puget Sound Enhancements	1	5	0	0	0	1	17
P16. Tactical Control System	7	2	0	0	6	0	28
P17. Tarrant County Integrated Justice Information System	0	0	0	0	3	1	3
P18. X-38 Fault Tolerant System Services	6	4	0	1	3	3	28
H01. CCHIT	4	0	0	2	1	1	35
H02. CM1	0	0	0	0	0	1	0
H03. InfusionPump	1	0	0	0	0	2	14
H04. Waterloo	7	20	0	4	4	5	53
C01. Amazing Lunch Indicator	1	0	0	0	0	1	12
C02. EU Rent	0	0	0	0	2	0	1
C03. FDP Expanded Clearinghouse Pilot	0	0	0	0	0	0	2
C04. Library System	0	0	0	1	0	0	6
C05. Nodes Portal Toolkit	4	0	0	0	0	1	8
C06. Online National Election Voting	3	0	1	0	0	0	10
C07. Restaurant Menu & Ordering System	2	2	0	0	0	0	2

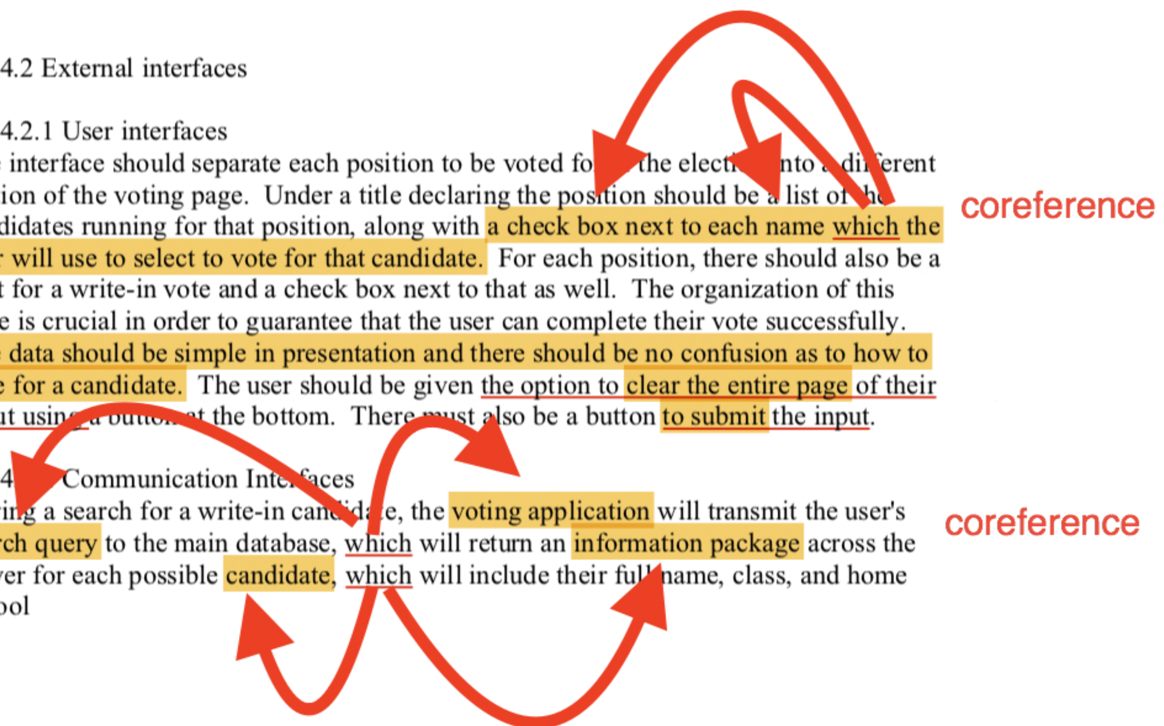
4.6 Problems of Semantic Sentence Ambiguity and Coreference

In requirements specifications, we expect that the text interpretation semantically corresponds to the *domain model*, i.e., the text makes sense in the context of the domain. Systems based only on parsing, as we discussed in Section 4.2.1 and Section 4.3, do not have access to the knowledge stored in the domain model required to make an interpretation properly.

We construct the model (as proposed in Section 3.4, in some aspects similar to the *UML class diagram*) from the textual requirements specification, but in this model, some interpretations need to be clarified. To solve this problem, we need a domain model and the part of the problem model that has already been built. Of course, at the starting point, the problem model is not available. Unfortunately, the domain model is often not available, too.

The only practical possibility is to build both these models incrementally and iterative during textual requirements processing through discussions with stakeholders. Then, suspicions of ambiguity can be generated by comparing the textual requirements specification and the model derived.

4.6.1 Linguistic Approach Completed by Knowledge Base



3.1.4.2 External interfaces

3.1.4.2.1 User interfaces

The interface should separate each position to be voted fo[illegible] the elect[illegible]nto [illegible] di[illegible]erent section of the voting page. Under a title declaring the position should be a list of [illegible] candidates running for that position, along with a check box next to each name which the user will use to select to vote for that candidate. For each position, there should also be a spot for a write-in vote and a check box next to that as well. The organization of this page is crucial in order to guarantee that the user can complete their vote successfully. The data should be simple in presentation and there should be no confusion as to how to vote for a candidate. The user should be given the option to clear the entire page of their input usin[illegible] the bottom. There [illegible] also be a button to submit the input.

3.1.4[illegible] Communication Interfaces

Duri[illegible] a search for a write-in can[illegible]e, the voting application will transmit the user's search query to the main database, which will return an information package across the server for each possible candidate, which will include their ful[illegible] name, class, and home school

Figure 4.13: Example of coreferential ambiguity.

Multiple coreference examples are shown in two extracted paragraphs from SRS [5] in Fig. 4.13. Let us inspect the sentence *"A check box next to each name which the user will use to select to vote for that candidate."* The determiner *which* could reference *a check box* or *name*. This one itself could indicate to think about the formulation. Nevertheless, there is a hint in the form of the *to select* verb. We can check the typical actions of both noun candidates using a knowledge base (e.g., ConceptNet [113]). If *select* is found as an action of a *check box*, our tool can hint a user with the probably intended candidate.

4.6.2 Model Approach

In the previous section, we used only the linguistic information. In this section, we include the information coming from the constructed model. This suggests a two-phase processing. In the first step, we use the linguistic information coming from the text, and we build the skeleton of the model. In the second phase, we analyze the text with regard to the obtained model. Because of that, we can find that two classes have the same attribute as you can see in this example: *"Our delivery contains product XYZ-123 in container X-50. Its weight is 50 kg."*

In the model, there are three classes: `Delivery`, `Product`, and `Container`. `Delivery` consists of `Product` and `Container`. The `weight` attribute candidate is not directly mentioned for any of the mentioned classes. Additionally, we may find that the attribute is common to all mentioned classes, e.g., in a semantic network. Therefore, the weight of 50 kg may concern any of them.

However, maybe the Product XYZ-123 is a radioactive isotope. In this case, its weight will not be 50 kg because, e.g., 235-uranium has 48 kg as its critical mass, so 50 kg would mean a nuclear explosion. It is surely obvious for all stakeholders, but it is a question of how obvious it is for the programmer.

4.6.3 Specific Attribute Values Distinguish the Coreference

In textual formulations, there may be a context (e.g., given by a coreference) in which the relation between an entity and the mentioned property represented by an attribute or its value is ambiguous. These ambiguities cannot be solved by syntactic means (like part-of-speech tagging) or by statistical means (like statistic sample data about co-occurrence).

To introduce the core of the problem, we present the following example sentence: *"In the picture, there is Lady Diana and her grandmother, as she was <number> years old."* We can see that changing the age number affects the coreference meaning of the pronoun *she*. Suppose there are some *OCL expressions* associated with these classes. In that case, we can use, e.g., *USE tool*[2] (UML-based Specification Environment)) to check which coreference option is consistent with the domain model.

[2] `https://www.db.informatik.uni-bremen.de/projects/USE-2.3.1`

Let us start with the model representation defined in the following Listing 4.3.

Listing 4.3: Model representation of a human (USE notation).

```
 1: model HUMAN
 2:
 3: class HUMAN_BEING
 4: attributes
 5:   age : Integer
 6:   sex : String
 7: end
 8:
 9: class MOTHER < HUMAN_BEING
10: end
11:
12: class GRANDMOTHER < MOTHER
13: end
14: association mother_of between
15:   HUMAN_BEING[1..*] role child
16:   MOTHER[0..1] role mother
17: end
18:
19: association grandmother_of between
20:   HUMAN_BEING[1..*] role grandchild
21:   GRANDMOTHER[0..2] role grandmother
22: end
```

We continue with the *OCL expressions* to support the previous UML model with the constraints in the following Listing 4.4.

Listing 4.4: OCL constraints of the model representing a human.

```
 1: context HUMAN_BEING inv: self.age >= 0
 2:
 3: context MOTHER inv MotherAge:
 4:               self.child->forAll(y|y.age < self.age - 12)
 5: context HUMAN_BEING inv MotherExclude: self.mother->excludes(self)
 6:
 7: context GRANDMOTHER inv GrandmotherAge:
 8:                    self.child->forAll(y|y.age < self.age - 24)
 9: context HUMAN_BEING inv GrandmotherExclude:
10:                    self.grandmother->excludes(self)
11: context HUMAN_BEING inv GrandmotherExclude2:
12:                    self.grandmother->excludes(mother)
```

Using this simple model, we can resolve the ambiguity for the number equals 18 of our example, i.e., that a grandmother cannot have 6 years, and the pronoun *she* is co-referencing Lady Diana.

In the case of the number equals 96, we can resolve the ambiguity too, because if Lady Diana were 96, her mother had to have at least 110 years and her grandmother at least 124 years, which is a possibility not contained in our realistic domain model.

In the case of the number equals 32, our domain model will not help us to solve the ambiguity shown above because both ladies can be 32 years old. In such a case, we have to generate a remark concerning the found ambiguity and generate a question for the analyst.

From the linguistic analysis, we have the information that Lady Diana is a woman (pronouns *she* and *her*), so we can set up the `sex` attribute of the object representing the `HUMAN_BEING` class from our domain model. We associate the found word *grandmother* with the class `GRANDMOTHER` in our domain model.

We are able to check defined *OCL expressions* with the mentioned *USE tool* – an example of the tool GUI is in Fig. 4.14.

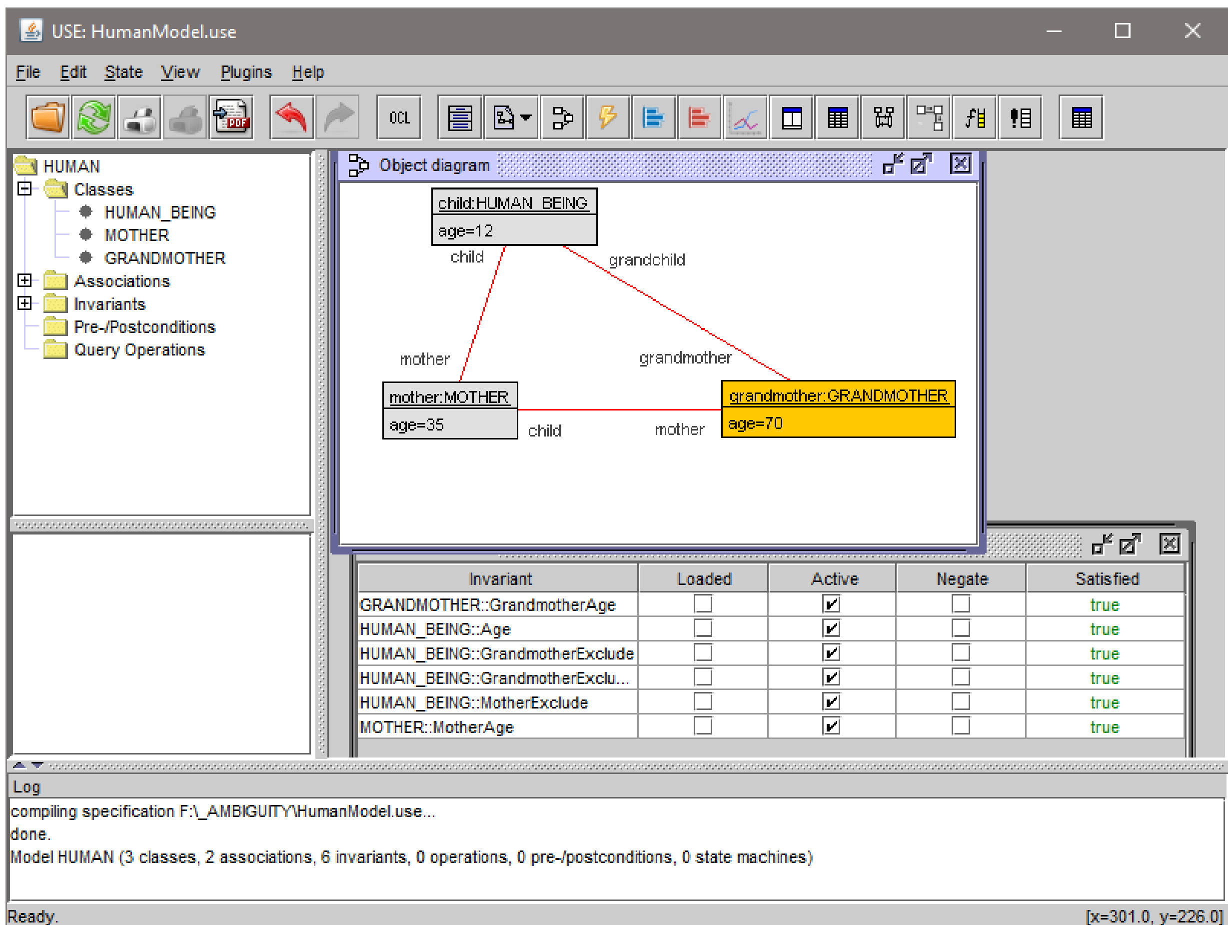


Figure 4.14: USE tool example.

Chapter 5

Problem of Incompleteness in Textual Requirements Specification

This chapter reflects our publications:

- Šenkýř, D.; Kroha, P. Patterns for Checking Incompleteness of Scenarios in Textual Requirements Specification. In: *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, Porto, 2020. [A.6]
- Šenkýř, D.; Kroha, P. Problem of Incompleteness in Textual Requirements Specification. In: *New Knowledge in Information Systems and Technologies*. SciTePress, Porto, 2019. [A.8]
- Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies*. SciTePress, Madeira, 2018. [A.10]

In this chapter, we investigate the incompleteness problem in textual requirements specifications. Incompleteness is a typical problem that arises when stakeholders (e.g., domain experts) hold some information for generally known, and they do not mention it to the analyst. A model based on the incomplete requirements suffers from missing objects, properties, or relationships as we show in an illustrative example.

We investigate this problem from two perspectives.

1. *A perspective of sentences and a created model. Our presented methods are based on grammatical inspection, semantic networks (ConceptNet and BabelNet), and preconfigured data from online dictionaries. Additionally, we show how a domain model has to be used to reveal some missing parts of it.*

2. *A perspective of scenarios and a created model.*

5.1 Problem Statement

According to [107], completeness means that all services required by the user should be defined. According to [43], a complete requirements specification has to cover all responses to all input data in all situations. Our ambitions are not so great. We try to reveal possible sources of incompleteness in the text of requirements.

Often, textual requirements do not contain complete information about the system to be constructed. There are more reasons for that.

First, we use a simplified model omitting some details of the real system to be modeled. We do not model and describe the complete reality, we are interested only in a subset of all existing objects, properties, and relationships the real system consists of. For example, when writing information systems, this subset is given by supposed queries needed to answer the user's requests.

Second, some details are forgotten initially, or some queries are appended later without worrying whether the underlying model contains the necessary objects, properties, or relationships.

Third, stakeholders working with the analyst on the textual version of requirements suppose that some facts are obvious and do not mention them. However, what is obvious for a user of a biological information system is usually not obvious for a computer scientist. We discuss such cases in Chapter 7.

Fourth, during requirements analysis, we can find some sorts of scenarios in functional requirements, e.g., normal case scenario (also called "sunny day scenario"), exceptional (alternative) case scenario (also called "rainy day scenario"), start-up scenario, shut-down scenario, installation scenario, configuration scenario, etc. Missing alternative scenarios are one of the incompleteness sources, i.e., descriptions of processing in the cases when something runs in another way than expected. It may easily happen. One of the cases is the following one. The analyst supposes a specific user's reaction in a given context because the analyst uses his/her knowledge of the problem. Contrary to the analyst's expectation,

the user does something else – it may be done by mistake or incompetence. In such a situation, alternative scenarios should guide the user to the next correct step.

Using incomplete information in requirements leads to incomplete or bad models. The resulting program has to be laboriously enhanced. Missing alternative scenarios can cause users to be disappointed and frustrated with the system, and it can cause customers to reject the system.

In [35], there is incompleteness together with ambiguity and other natural language processing defects called *requirements smells*, similar to the concept of *code smells*.

To illustrate the problem of incompleteness, we introduce two typical examples. In Example 5.1, the source of incompleteness is that some attributes of the class `Restaurant` are not clearly and uniquely defined.

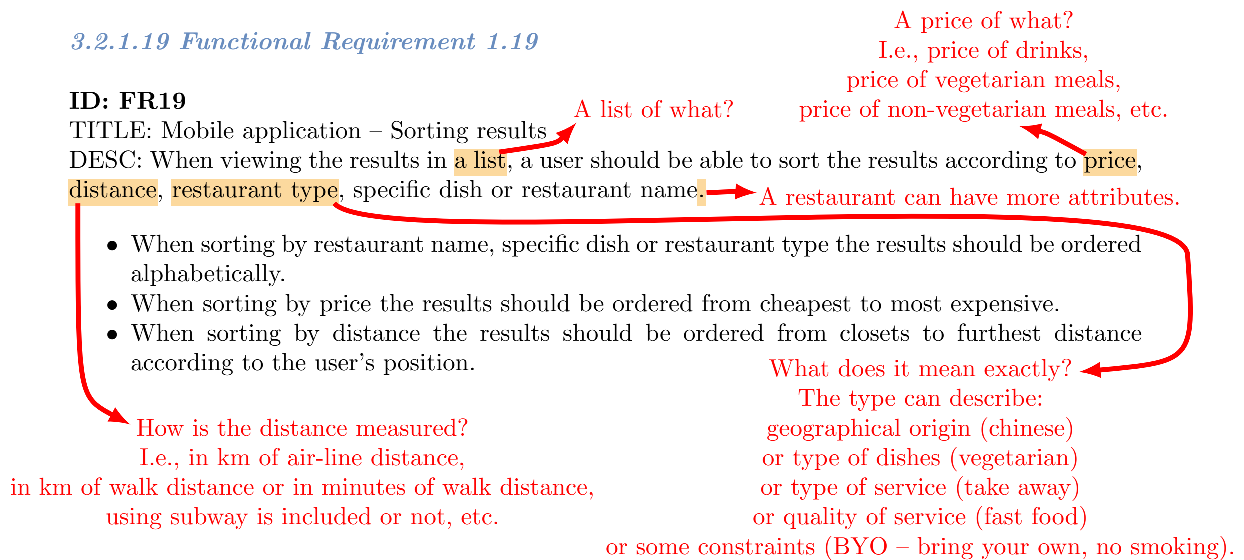


Figure 5.1: Example of incompleteness (restaurant).

Example 5.1: Restaurants.

In the description of *functional requirement Nr. 1.19* in Fig. 5.1, taken from [47] that is part of our collected data set (Section 3.6.1), we can see some incomplete formulations. The user should have the possibility to sort the list of found restaurants according to attributes that are not uniquely defined. We extracted the following sources of incompleteness:

1. ...viewing the results as a list ⟶ a list of what?
2. ...a user should be able to sort the results according to price ⟶ the price of what (price of drinks, price of vegetarian meals, price of non-vegetarian meals, etc.)?

3. ...distance ⟶ how is the distance measured, i.e., in km of air-line distance, in km of walk distance, or in minutes of walk distance using the subway included?

4. ...restaurant type ⟶ what does it mean exactly, i.e., the type can describe, e.g., a geographical origin (Chinese), a type of dishes (vegetarian), a type of service (take away), a quality of service (fast food), some other constraint (BYO = bring your own, no-smoking, etc.).

In Example 5.2, the set of queries used by the information system of a military university contains the attribute `shoe size`, but the set of queries used by a university of technology does not.

Example 5.2: University information system.

Suppose we want to write requirements specifications for a university information system. Modeling attributes of classes `Student` and `Teacher`, we need to include attributes like `first name`, `last name`, `address`, etc.

However, we do not include the attribute `shoe size` in the case of a university of technology, but we have to include it in the case of a military university because of a uniform. This means that the completeness of the set of attributes of the class `Student` can be decided only with respect to the set of queries or operations that concern the data stored.

5.2 Related Work

Completeness of requirements specification is a complex problem discussed for years in detail in many publications, e.g., in [41] and [35].

In this section, we discuss some key areas and approaches related to the research of software-assisted detection of incompleteness in textual requirements.

5.2.1 Incompleteness Detection Tools

A tool called *Cordula* (Compensation of Requirements Descriptions Using Linguistic Analysis), presented in [11], follows the vision of *On-the-Fly (OTF) Computing*. This vision assumes ad hoc providing software services based on requirements description written by a user in natural language. The authors of the paper discuss the question "How users might be involved in the compensation process?". *Cordula* represents a chatbot [45]. The chatbot can highlight the inaccuracies, and a user can confirm the proposed suggestion by the chatbot or reject it. The proposed approach engages the users in the following steps. First, they describe their requirements in a short text. Based on that, they receive a response from the chatbot. Second, they are asked to react to the proposed suggestion. Based on that, the chatbot can adapt to the situation.

In the prerequisite paper [10], there are presented methods of quality violations in a requirements specification. The authors call them *linguistic triggers*. Besides the problem of incompleteness, there is also presented an approach to ambiguity detection. In the case of incompleteness triggers, authors pursue an already presented approach divided into two steps [9]. First, the detection via *predicate-argument analysis* in which a *semantic role labeler (SRL)* assigns semantic roles such as *agent*, *theme*, and *beneficiary* to the recognized predicate. The presented illustrative example is the verb "send", which is a three-place predicate because it requires the agent (sender), the theme (sent), and the beneficiary argument (sent-to). If the beneficiary is not specified here, it is unknown whether one or more recipients are possible. The second step is compensation. The authors state that they gathered software descriptions and their corresponding reviews from the site `https://download.cnet.com`. Using the *similarity search component* known from the *information retrieval* (IR) domain, they try to find the potentially missing part *sent-to*.

The authors of [31] focused on *performance requirements* – the requirements that describe system behavior. They created a UML model reflecting three categories of performance requirements – *time behavior* requirements (e.g., "The operation must have an average response time of less than 5 seconds."), *throughput* requirements (e.g., "The system must have a processing speed of 100 requests/second."), and *capacity* requirements (e.g., "The system must support at least 50 concurrent users."). Following the UML model, they are able to map a textual requirement to parts of the model. Based on the predefined mandatoriness of each single part, they can indicate the possible missing parts. The mapping of textual requirements is done via *sentence patterns* derived from the model.

A tool called *NLARE* (A Natural Language Processing Tool for Automatic Requirements) is presented in the papers [60] and [59]. There is a functional requirement defined as a finite set of words where words can form one of 3 three elements: *actor* (performs the action in the statement), *function* (indicates what action needs to be performed), or *detail* (indicates conditions under what action be expected). Based on this categorization, incompleteness exists if any of the mentioned elements is not found. Mapping of words and elements uses patterns represented by regular expressions that operate on part-of-speech tags.

The work of Chattopadhyay et al. [19] focuses on user stories of one project and feature descriptions of another project. On the level of sentences, they identify verbs (using part-of-speech tagging), and they try to find all typical structure usages of identified verbs defined in FrameNet[1]. Based on the proposed structure usage, they check if this structure is used in the text of a requirement. Our approach differs in using semantic networks, and we also do not focus only on verbs.

5.2.2 Incompleteness Confrontation

The research in the corresponding area of processing of textual corpora also takes into the problem of incompleteness. For example, in [98], authors propose methods of knowledge

[1] `https://framenet.icsi.berkeley.edu`

acquisition from Wikipedia, and they argue that it should be a source of minimization of incompleteness.

Paper [35] classifies the problem of incompleteness as one of the *requirements smells*. Following ISO 29148 [65] requirements engineering standard, they introduce a category of *requirements smell* called *incomplete references* that should be processed in their tool *Smella*. The detection mechanism is not in the scope of their paper.

Also, a paper with a concise name "What Did You Mean" [48] refers to the problem of incompleteness. They follow [33] where the incompleteness is indicated by *under-specification*. An under-specification indicator is pointed out in a sentence when the subject of the sentence contains a word identifying a class of objects without a modifier specifying an instance of this class. Concerning the example sentence – "The system shall be able to run also in case of attack." – we ask which attack the writer means?

5.2.3 Incompleteness of Scenarios

In [116], scenarios and their usages are described in detail, but their completeness or incompleteness is not mentioned.

Two incompleteness metrics of input documents of the requirements specifications are described in [38]. This approach takes into account all the relevant terms and all the relevant relationships among them, and it defines forward functional completeness and backward functional completeness. The forward functional completeness corresponds with the reference functional model, i.e., with the future implementation of the system. The backward functional completeness, which the paper focuses on, refers to the completeness of a functional requirements specification with respect to the input documents.

In our approach, we mix both methods. We use requirements specifications first to build a model that can be implemented (as we described in Section 3.4) and check in the sense of the forward functional completeness (the methods presented in Section 5.3 except the scenario-based method). Then, in the next step, we use the input documents, the existing text of requirements specification, the fresh model, and some information from external knowledge databases to search for the backward functional incompleteness that is represented by missing alternative scenarios (Section 5.4). Contrary to the approach in [38], we do not measure the incompleteness using metrics and quantified results, even though it is a good idea. Using our tool, we generate warning messages only.

In [81], a meta-model approach is used to detect the missing information in a conceptual model. It is also an approach of the class forward functional completeness but at the level of a conceptual model.

In [31], sentence patterns are used to uncover incompleteness with performance requirements. According to the unified model, the performance requirements describe time behavior, throughput capacity, and cross-cutting. The sentence patterns used in the paper are completely different from our sentence patterns.

In [25], the authors explore potential ambiguity and incompleteness based on the terminology used in different viewpoints. They combine the possibilities of NLP technology with information visualization. Their approach is entirely different from our approach.

The approach based on NLP techniques and grammatical inspection methods to extract use case scenarios is proposed in [122]. It is a similar approach to ours, but the question of incompleteness is not discussed there.

5.2.4 Related Works – Overview

The presented papers use, similarly to us, methods of natural language processing in the sense of resolving *part-of-speech tags* and the *dependency structure* of a sentence. Our work differs in the usage of support resources. In Section 5.3, we present the sources that we use – a prepared static collection of collocations and online resources: *ConceptNet* and *BabelNet*. Our implemented tool analyzes the whole document at once and generates warnings. This is the difference compared with the mentioned chatbot in [45].

5.3 Our Approach

The goal of our project is to identify sources of incompleteness in textual requirements. In our approach, we distinguish the sort of incompleteness according to the scope, in which the incomplete information is spread in the text of requirements specification. The first scope includes one sentence of the textual requirement specifications. The second scope includes the whole document of the textual requirements specifications, i.e., all its sentences.

We follow with a categorization of our investigated methods.

- Scope of sentence (S):
  - the usual usage of words – group S.1,
  - acronyms definition – group S.2.
- Scope of the whole document (D):
  - the semantic knowledge stored in our model – group D.1,
  - the information contained in the set of proposed queries as sources of actions to be performed – group D.2,
  - the draft of the pre-generated model – group D.3,
  - the proposed scenarios – group D.4.

Our approach to the problem of incompleteness uses grammatical patterns based on grammatical inspection, semantic networks such as *ConceptNet* [113] and *BabelNet* [86], and pre-configured data from online dictionaries. We also show how a domain model can be used to reveal a possible problem of a single incomplete requirement (i.e., missing details) and to reveal a possible problem of incomplete requirements specification as a whole.

Our tool TEMOS generates warning messages in the case when a suspicious formulation has been found. A warning message signals the necessity of human intervention.

5.3.1 Group S.1 (Usual Usage of Words)

Nouns in sentences are investigated using a specific, predefined set of words. If the kind of usage in the textual requirements (e.g., *a list*) does not correspond to any defined usages in the vocabulary (e.g., *a list of*) then a warning message is generated.

The set of words we used for this purpose is available on our web page[2], hereinafter referred to as a *common noun and preposition collocation set*. We build this collection based on various collocation web pages. Each entry is provided with a resource reference. We would like to mention *Free Online Collocations Dictionary*[3] from *ProWritingAid*. The advantage of this dictionary is that one can find collocation also via preposition (e.g., *of*). Based on the query, the result contains collections of commonly used collocated words grouped by the *part-of-speech* category. Our second significant resource is *Corpus of Contemporary American English*[4] (COCA). One can search the corpus via expression containing *part-of-speech* tags. In our situation, we were looking for common collocation, e.g., of preposition *of*. Therefore, we inserted the `[nn*] of` query expression. The result list contains found words sorted by the frequency of usage.

The search for nouns belonging to this group S.1 can be done using a simple sentence pattern presented in Fig. 5.2 that uses corresponding part-of-speech tags and the `preposition` dependency type relation.

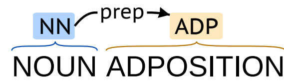


Figure 5.2: Noun with preposition pattern.

If this pattern is matched, then no further action is needed. Otherwise, we will look at the mentioned preconfigured *common noun and preposition collocation set* to check if the tested noun is to be found together with a preposition or not. Because the concrete noun can have various forms, we check the *lemma* (dictionary form) of the noun.

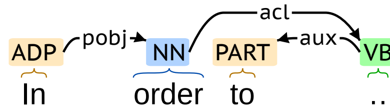


Figure 5.3: "In order to action" pattern.

To precise this method, in the case when the simple pattern is not matched, we use the whitelist of supporting patterns. The purpose of these patterns is that, during the testing phase, we indicate some repetitive parts of sentences that should not be indicated as a warning. For example, we can show the whitelist pattern of common sentence start illustrated in Fig. 5.3. When any of the whitelist patterns are matched, no warning is generated, and the analysis continues with the next word.

[2] https://temos.ccmi.fit.cvut.cz/sources
[3] https://prowritingaid.com/en/Collocation/Dictionary
[4] https://www.english-corpora.org/coca

5.3.2 Group S.2 (Acronyms Definition)

Here, we reuse the concept of glossary discussed in the previous chapter concerning ambiguity. If we find a word composed only of capital letters, we consider it as an acronym and check whether the acronym is defined in the glossary.

This approach can be refined by using a configurable word length limit. The reason is that some writers use capitals to highlight a word. When the word length limit is set, e.g., we consider a word as an acronym up to 5 characters, then no (probably) false warning is generated for longer words.

5.3.3 Group D.1 (Semantic Knowledge)

The semantic knowledge stored in our model of reality contains some specific information, e.g., the information that not every restaurant has to serve meat.

In the textual expression, this kind of information corresponds with a possible (but avoided) existence of:

- adjectives before nouns, e.g., *a vegetarian restaurant*, or
- compound nouns, e.g., *a university restaurant.*

The crucial nouns to check are those nouns that represent entity candidates. Let us define a set of nouns N_e where each noun $n \in N_e$:

- represents a lemma of an entity candidate according to our internal model definition (Section 3.4),
- there are missing adjectives before noun n, and noun n is not part of compound nouns.

We iterate over all $n \in N_e$, and we check the following information sources.

Our first source of information is *ConceptNet* [113], which provides the *is-a* relation (an example of the endpoint call for the noun *restaurant* is in Listing 5.1). We check the found collocations (e.g., *French restaurant*) of the queried noun n, and we recommend it to a user. In the result, there could also be terms that are not collocation of the queried noun, e.g., *bistro is a restaurant*; however, we can display them to a user as hints, too.

Listing 5.1: ConceptNet – is-a relation – call the endpoint.

```
GET
https://api.conceptnet.io/query
   ?node=/c/en/restaurant
   &rel=/r/IsA
```

Concerning our example of restaurants, if we query the term *restaurant* directly in the ConceptNet web application GUI, the *is-a* relation is presented as *types of restaurants.*

Our second source is *BabelNet* [86], which provides the *has kind* relation. Similarly to the previous, if some examples are found for a specific noun n, we generate a question concerning the possibility of missed specialization of the entity.

Another aspect of this group (D.1) targets *element groups* (as defined in our internal model). *Element group* clusters elements with the same root lemma, i.e., a lemma of the main word (token) representing the element. It could happen that an element group consists of elements with adjectives or compound nouns and one element that is represented by the root lemma only. In our example context, we can have a *restaurant* and a *vegetarian restaurant* as elements in one element group. Does it mean that all occurrences of the word *restaurant* in the text of requirements represent a *vegetarian restaurant*? We can generate a question for a user regarding that.

5.3.4 Group D.2 (Actions)

Verbs in sentences of queries are investigated in the sense of whether the action that they describe can be performed in the existing model (e.g., sorting without a key, sorting without a unique key, the number of shoes of size 42 for students of military university).

As a prerequisite, we are able to check the relationships in the way of correct usage of the verb. The English verbs can take 0, 1, or 2 (direct and indirect) objects, depending on the verb. Verbs without objects are called *intransitive*, and the other ones are called *transitive*. Using the dependencies recognition, we check if the verb has any objects. If no object is found, we check the verb against the list of intransitive verbs (e.g., Wiktionary collection of such verbs[5]). For example, the standalone sentence *"A warning appeared."* does not bring new information, but it is grammatically correct. On the contrary, the standalone sentence *"Administrator needs to maintain."* contains transitive verb *need*, and we are missing the information about what needs to be maintained. Therefore, such a sentence is suspicious, and TEMOS generates a warning for the user.

5.3.5 Group D.3 (Model Validation)

We process each sentence one by one and incrementally build our internal model in the sense of the UML class model as proposed in Section 3.4.

When the draft of our model is ready, we have a chance to check the following simple indicators of missing or unrecognized information:

- *"empty" classes* (entity candidates) without attributes,
- classes (entity candidates) with *no relationship* to any other class.

[5] `https://en.wiktionary.org/wiki/Category:English_intransitive_verbs`

5.4 Our Approach – Incompleteness of Scenarios

For examples of group D.4 (the proposed scenarios), we set aside this separate section.

As mentioned, we can find some sorts of scenarios in functional requirements, e.g., normal case scenario (also called "sunny day scenario"), exceptional (alternative) case scenario (also called "rainy day scenario"), start-up scenario, shut-down scenario, installation scenario, configuration scenario, etc. Missing alternative scenarios are one of the incompleteness sources, i.e., descriptions of processing in the cases when something runs in another way than expected.

The goal of our method described in this section is to find such a kind of incompleteness that can be revealed only in the context of the whole textual document or in the context of thematically closed and compact sections.

In this section, we present some simple examples that concern user interface. As the norms *ISO/IEC/IEEE 29148-2018* [66] and *IEEE 1012-2016* [63] explain, the specification of a user interface is a part of the software requirements specification.

5.4.1 Alternative Scenarios

More or less, we are trying to reveal the non-existence of some alternative scenarios. As an alternative scenario, we denote here a scenario that depends on a specific value of a class attribute.

Scenarios are used for requirements specification elicitation. We use three types of scenarios in our approach:

- a description of the system context (input events, system output, i.e., the system's communication with the actors out of the system),
- a description of system usage, including users' goals (including use cases) and system function,
- a description of constraints that concern attributes or application of methods.

We assume that some alternative scenarios have to be present in the textual description of the requirements specification.

First, we state which alternative scenarios should be present, i.e., we construct a set of alternative scenarios for each normal case scenario.

Second, we find the alternative scenarios present in the textual requirements, and we compare the corresponding sets.

Let us continue with the following illustrative example. Assume we have textual requirements specification of a text editor. One of the use cases is described as a functional requirement, as proposed in Example 5.3.

Example 5.3: Text editor.

Normal case scenario: To edit a text file, the user has to click the *Open* button to choose the file he/she wants to edit. After the user sees the file content, he/she provides the changes to its textual content. To save the changes, the user uses the *Save* button.

Alternative (exceptional) scenario 1: . . . If the user has got the message "The file cannot be found", the user has to do the following. . .

Alternative (exceptional) scenario 2: . . . If the user has got the message "The file you want open is not a text file", the user has to do the following. . .

To have the possibility of testing the existence of these alternative scenarios, we need a class `File` containing attributes `status` (values: `exists`, `not found`, `opened`, etc.) and `type` (values: `docx`, `doc`, `rft`, `txt`, `tex`, etc.) in the model. Some text editors can open files of types such as `PDF` or `PNG`, but the user can be confused by the result he/she can see on the screen.

In some cases, the implementation will be written as an exception-handling procedure that delivers a message the user can understand instead of the message of the operating system.

In *Adobe Photoshop CC*, you will get the message *"Could not complete your request because Photoshop does not recognize this type of file"* if you try opening a file of type `DOCX`.

In *Microsoft Word*, you can try opening a `PDF` file, but you will get the message *"To open and export to certain types of files, Word needs to convert the file using a Microsoft online service"* in the first step.

If our model (constructed via extracting classes and attributes) has the property described above, we can check whether the corresponding alternative scenarios are part of the textual requirements specification.

5.4.2 The Algorithm

The algorithm implemented in our tool follows these steps:

Algorithm 5.1: Revealing alternative scenarios.

1. To identify enumerations, paragraphs, and chapters by using white and special characters.

2. To construct a static UML model by using grammatical inspection, i.e., to find classes, relationships, and attributes. Section 5.4.3 provides additional informa-

tion.

3. To find sets of values of each attribute as described in Section 5.4.4.
4. To find alternative scenarios in all components (chapters) of the specification, i.e., such sentences that use different values of the same attribute. A component of a specification is the basic part of the structured text of the specification. Usually, components are numbered, as we show in Example 5.6. This core step of the whole algorithm is described separately in Section 5.4.5.
5. To find groups of attribute values because some attribute values can be grouped together, and they can use a common alternative scenario. Such a group of attribute values has to be identified.
6. To test whether there are alternative scenarios for all attribute values described in all components.
7. To generate warning messages if some alternative scenarios are missing.

5.4.3 Static UML Model Construction

Using grammatical inspection, our tool TEMOS finds classes, their attributes, and the constraints involved, as we described in Chapter 3 (Overview of Our Approach). When we have classes and their attributes, our tool looks for their values, and it builds a corresponding set of values to each attribute of each class that participates in *scenarios*.

These sets need not be built only from the text of requirement specifications. Using some pre-defined knowledge databases, we can generate warning messages, e.g., we find in a knowledge database that the `File` class (from Example 5.3) can have a `status` value `locked` or `encoded`. Such a value is not mentioned in the requirements specification. We suppose that the alternative scenario concerning the case of a locked or encoded file is missing, and we indicate incompleteness.

5.4.4 Sets of Values

In common, we can describe our approach as follows. We denote sets of attribute values that are built from the textual requirement specification as R-sets having cardinalities $Card(R\text{-sets})$ and sets of attribute values that are built from scenarios as S-sets having cardinalities $Card(S\text{-sets})$.

As a result, each attribute $Attr$ of each class C has a corresponding set of its values taken from requirement specifications denoted as $R_{C,Attr}$, and each of these sets has its cardinality $Card_{R,C,Attr}$. Similarly, each attribute $Attr$ of each class C has a corresponding set of its values taken from scenarios denoted as $S_{C,Attr}$, and each of these sets has its cardinality $Card_{S,C,Attr}$.

In the first step – we can call it calibration – the cardinalities of sets constructed from requirements will be compared with the cardinalities of sets from scenarios. Probably, it will be found that $Card_{R,C,Attr} > Card_{S,C,Attr}$. If more values are mentioned in the specification than in the scenarios, it means that scenarios do not cover all possible situations or some values need not be taken into account. Our tool will generate a message to check this situation.

The case in which $Card_{R,C,Attr} < Card_{S,C,Attr}$ means that the information obtained from requirements contains fewer attribute values than the information obtained from scenarios. It happened in cases when requirements do not count with all values, maybe because all values were accumulated at the time of writing the scenarios. Our tool will generate a message, too.

The case in which $Card_{R,C,Attr} = Card_{S,C,Attr}$ means that we can start the incompleteness checking in the scope of the whole document as follows.

In the second step, we suppose that the reason why a set of attribute values is mentioned in the description is that each of these values indicates a different path of data processing. Theoretically, we could expect alternative scenarios for different attribute values, as shown in the following example.

Example 5.4: Traffic simulation.

In the traffic simulation, the class representing `Traffic Light` has the corresponding set of attributes such as `type`, `year of production`, `light` (representing the current light that is active), etc. From the specification, we know the possible values of the attribute `light`. It is this set: `red`, `orange`, and `green`.

Scenario for "light" equals "red":
The stop procedure is applied...

Scenario for "light" equals "green":
The run procedure is applied...

The enumeration of attribute `light` was ended and not all possible values were mentioned. Our tool generates a warning message: "What is to do if the "light" is "orange", i.e., the alternative for "light" equals "orange" is missing?"

In practice, some attribute values can be grouped, and the alternative scenario is defined for the whole group. It means that multiple values are mentioned together and separated by commas, or the multiple values are hidden behind the noun *"others"*.

5.4.5 Patterns To Find Scenarios

Following Algorithm 5.1 described above, we replace the problem of finding scenarios with the problem of finding typical textual patterns that indicate the presence of scenarios in a text of requirements specification. We reuse the already presented idea of the *grammatical inspection* and *sentence patterns*, as discussed in Chapter 3 (Overview of Our Approach). So, we construct the corresponding patterns, we apply them to the whole text, and we find all sentences in which such a situation can be found.

We can categorize the patterns into two groups. The patterns from the first group (Section 5.4.5.1 and Section 5.4.5.2) cover situations where the class name and names of its attributes have to be mentioned. The patterns from the second group (Section 5.4.5.3) cover situations where the values of some attribute are used but the attribute name is not explicitly mentioned.

5.4.5.1 Patterns Covering Class/Attribute Name

Usually, the class name and its attributes are embedded in a description of a process started by calling a method. Often, there is the same verb in a negative clause in the alternative scenario. The example situation follows.

Example 5.5: File processing.

The `File` class has methods `Open` (means open existing file), `Enter data` (or `Import` file from an external device), `Save` (without changing attributes `name`, `directory`, or `type`), and `Save As` (changes are possible).

Normal case scenario:
To process an existing file that **can be seen** in the window of the file manager, the user uses the "Open" item in the "File" menu.

Alternative scenario:
To process an existing file that **cannot be seen** in the window of the file manager, the user uses the "Import" item in the "File" menu.

Normal case scenario:
After the action "Enter data" is completed, and if the data **is** ok, the system shall store the data.

Alternative scenario:
After the action "Enter data" is completed, and the data **is not** ok, the system shall issue an error message.

Based on the proposal, we introduce two pattern categories here. The first category represents the *values conditional pattern* (variants are in Fig. 5.4 and Fig. 5.5), which

checks if the requirements enumerate behavior based on all mapped values of a concrete attribute. If there is a scenario for at least one concrete value of a specific attribute, it should be checked if all already mapped values of this specific attribute are concerned.

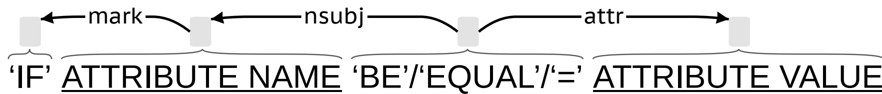


Figure 5.4: Values conditional pattern #1.

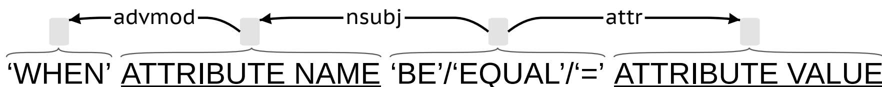


Figure 5.5: Values conditional pattern #2.

The second category represents the *incomplete conditional pattern*. Typically, the requirement describes the prospective situation in a conditional way (e.g., if something is (successfully) loaded/processed/OK), but alternatives are missing. Based on this observation, we can create two sets. The first one contains prospective nouns (Fig. 5.6) and verbs (in the *past participle* ("-ed") form, Fig. 5.7), and the second one contains their opposites. When the scenario mapping prospective situation appears, it is time to check if the opposite exists. The not resolved challenge represents the formulations that are not directly negation, e.g., the next statement starts with the word *"otherwise"*.

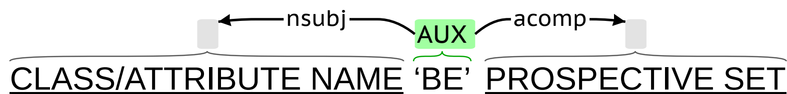


Figure 5.6: Incomplete conditional pattern #1.

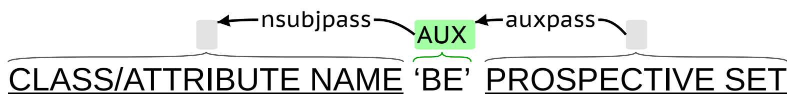


Figure 5.7: Incomplete conditional pattern #2.

5.4.5.2 Unique Attribute Pattern

In the requirements, some attributes may be marked as unique (as presented in Section 3.4). When such a unique attribute is used within the extracted positive or negative use case scenario, an alternative scenario should indeed exist. We can find it via the pattern in Fig. 5.8, where the *unique set* should consist of words such as *taken*, *occupied*, *used*, etc. Attribute representing *"username"* is a typical example. When a user tries to register an already-taken username, we need an alternative scenario for such a situation.

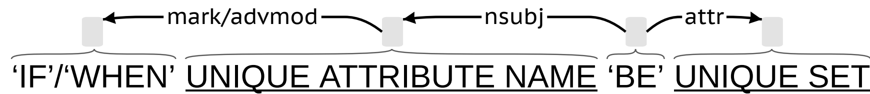


Figure 5.8: Unique attribute pattern.

5.4.5.3 Patterns Covering Values of Attribute

In this section, we focus on sentences where the name of the attribute is not mentioned.

The *values in enumeration pattern* considers enumerations, and the pattern is based on the text structure of a block of requirement statement(s). Different from the *values conditional pattern*, we do not require a conditional form in this case.

As shown in Example 5.6[6], adapted from our old faculty guideline page, all (three) points of enumeration listed in the *Installation* part together cover all values of one specific attribute (representing *operating system*). However, only two of them are discussed in the *Configuration* part.

Example 5.6: OpenVPN.

Non-functional requirement specification 5.13: The system OpenVPN will work under Windows, macOS, and Linux.

Design specification 5.13:
5.13.1 Installation of OpenVPN
We need an OpenVPN client from the community edition at least of version 2.4.

- **Users of MS Windows** should download from https://openvpn.net/index.php/open-source/downloads.html
- **Users of Linux** will very probably use a package from their Linux distribution. It can be useful to install Network Manager, too.
- **Users of macOS** have the application available from https://tunnelblick.net, which contains both Open VPN and the graphical interface.

5.13.2 Configuration of OpenVPN

- **Users of MS Windows** are recommended to store configuration files in their personal profile, e.g., in folder `C:\Users\Name\OpenVPN\config` where `Name` is the user's name in MS Windows.

[6]This example is taken from a grey zone between non-functional requirement specifications and a design specification. However, the border between requirement specifications and the design specification is not exactly defined, and usually, they overlay each other in some aspects and properties.

- **Users of Linux** place the configuration file into the folder that is used by their specific distribution of Linux, probably \etc\openvpn.

To configure OpenVPN we need the following files...

One can see that the name of the attribute is not explicitly mentioned there. Therefore, the first step of mapping this pattern is to check each item of enumeration against the values of all attributes. By this step, we find the attributes that are possibly enumerated, and we can check the missing values. See illustrative Fig. 5.9.

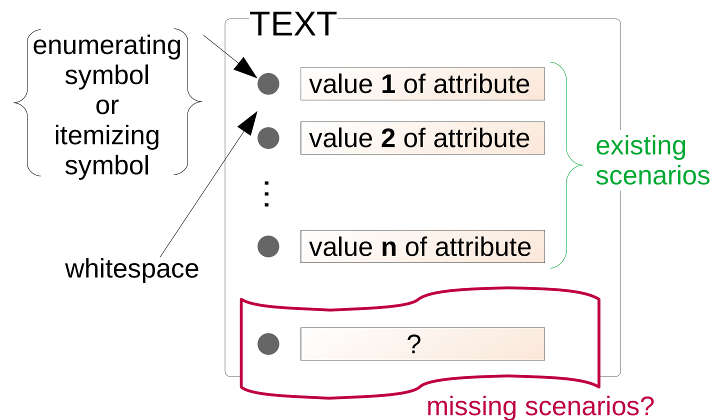


Figure 5.9: Values in enumeration – check approach.

This kind of incompleteness has to be found and indicated by our tool, i.e., a message like *"In the 5.13.2 Configuration part of the specification, the alternative concerning macOS is missing."* has to be generated.

5.5 Experiments and Results

In this section, we first present two illustrative examples from one selected software requirements specification where our methods could help. The software requirements specification is called *Amazing Lunch Indicator* [47], and it is part of our collected data set (Section 3.6.1).

Second, we present an evaluation of our proposed methods using the entire dataset (Section 3.6.1) in the form of the frequency of matched patterns.

5.5.1 Evaluation Example #1

We continue in the analysis of Example 5.1 (description of functional requirement Nr. 1.19) from this chapter's introduction – Section 5.1 (Problem Statement).

Our tool TEMOS delivers the first version of classes, attributes, and relationships, including the class representing restaurant – class `Restaurant` having attributes: `name`, `telephone number`, `type of food`, `average price`, `description`, `link` (to the

restaurant's web page). The connected `Restaurant element` class used in the listing of restaurants has attribute `distance` (according to the user's position).

The procedure of analyzing the text of *DESC* (authors use it as a shortcut of *description*; it does not mean *descending sort order*) in *FR19* will run as follows.

1. The Oxford Advanced Learner's Dictionary[7] defines the noun *"list"* as *a series* ***of*** *names, items, figures, etc.* According to the definition, this means that the "... **of what**" part may have been omitted (group S.1, Section 5.3.1).

   - Our tool TEMOS generates the first warning message: *"FR 19 – DESC: A list* ***of what*** *should be generated?"*
   - After a discussion with stakeholders, the analyst writes a new formulation: *"DESC: When viewing the results in a list* ***of restaurants****, a user should be able to sort the* ***restaurants*** *according to price, ... "*

2. As the updated sentence analysis continues – *"... to sort the restaurants according to price"* (assumption: the existence of a unique sort key among attributes of objects to be sorted, i.e., of the corresponding class), our tool TEMOS checks whether the `Restaurant` class has the `price` attribute that can be used for sorting – information of the group D.2 (Section 5.3.4). The exactly named attribute `price` is among the attributes of the `Dish` class, but it is not among the attributes of the `Restaurant` class.

   - As it is not the case, TEMOS generates the following warning message: *"Restaurants as results of the search cannot be sorted according to the 'price' attribute because 'price' is not an attribute of the 'Restaurant' class. The 'Restaurant' class has the following attributes: name, telephone number, type of food, average price, description, and link. Do you mean the 'average price' attribute?"*

The situation is even more complicated if we should sort restaurants according to their distance from the user's position. Without sorting, it would be enough to show a map to the user with his/her position and the position of the restaurant. Unfortunately, this cannot be used for sorting.

If we were interested in air-line distance, we would need the GPS coordinates of the restaurants, which would belong to attributes of the `Restaurant` class, and we need to obtain the GPS coordinates of the user's position. However, the `Restaurant` class does not have such an attribute mentioned in the requirements. The `distance` attribute is mentioned in the connected `Restaurant element` class, but how it would be computed is not mentioned. Moreover, the air-line distance is not very useful in many towns. Often, it is much more important to know how much time we need to achieve the goal place. So, we speak about distance, but we mean the time interval. Other metrics, e.g., Manhattan

[7] `https://www.oxfordlearnersdictionaries.com/definition/english/list_1`

metric, using a subway or a taxi route, are complicated, too. As we can see, a simple formulation of requirements can cause implementation problems. The description in detail is out of the scope of this investigation.

5.5.2 Evaluation Example #2

We continue with the same software requirements specification as in the previous section. This time, we investigate the *functional requirement Nr. 1.6* stating *"The search options are price, destination, restaurant type, and specific dish."* Our tool maps these four options as values of the `search option` attribute.

The *functional requirement Nr. 1.8* contains the following rule: *"When searching by a search option other than price, the restaurants should be sorted according to the following order: 1. distance, 2. average price, 3. restaurant type, 4. specific dish."* The order represents a list where 3 of 4 values of the `search option` attribute are mentioned. The *values in enumeration pattern* (Section 5.4.5.3) from the group D.4 is applied in such a case. Therefore, our tool generates a warning regarding the missing value *destination*. Based on this warning, the analyst can decide whether the distance is an expression of the destination in this context, and if the meaning is clear.

5.5.3 Results

Table 5.1 summarizes the results of experiments of our presented incompleteness detection approach using the data set introduced in Section 3.6.1. The results include word count (for comparison with group S.1 results) and the frequency of matched patterns of groups S.1, D.1, D.2, D.3 (where we focus on classes (entity candidates) with no relationship to any other class), and D.4.

Table 5.1: Evaluation of methods detecting incompleteness.

Case	Words	S.1	D.1	D.2	D.3	D.4
g02-federalspending	2,089	7	8	0	0	0
g03-loudoun	1,579	30	9	0	0	0
g04-recycling	1,287	11	7	0	0	0
g05-openspending	1,640	3	6	0	0	0
g08-frictionless	1,746	29	8	0	0	0
g10-scrumalliance	2,571	36	16	0	0	0
g11-nsf	1,749	10	6	1	0	0
g12-camperplus	1,397	22	7	0	0	0
g13-planningpoker	1,457	21	10	0	0	1
g14-datahub	1,841	13	6	0	0	0
g16-mis	1,536	18	11	0	0	0
g17-cask	1,627	4	5	0	0	2
g18-neurohub	2,200	25	15	0	0	0
g19-alfred	2,441	16	11	0	0	3
g21-badcamp	1,889	14	6	1	0	2
g22-rdadmp	2,246	20	7	0	0	1
g23-archivesspace	875	6	3	0	0	3
g24-unibath	1,464	7	4	0	1	0
g25-duraspace	2,015	15	11	0	1	0
g26-racdam	2,122	17	8	0	0	2
g27-culrepo	3,316	39	19	0	0	3
g28-zooniverse	1,060	4	6	0	0	0
P01. Blit	535	5	5	0	0	0
P02. CS179G – ABC Paint Project	1,199	7	10	1	1	0
P03. eProcurement	1,683	35	19	0	0	0
P04. Grid 3D	196	1	4	0	0	0
P05. Home 1.3	1,121	13	11	0	0	0
P06. Integrated Library System	1,974	84	34	0	0	2

Case	Words	S.1	D.1	D.2	D.3	D.4
P07. Inventory	4,657	71	24	4	0	5
P08. KeePass Password Safe	466	4	7	1	0	0
P09. Mashbot	619	6	7	0	0	0
P10. MultiMahjong	1,759	32	17	0	0	2
P11. Nenios	944	53	10	0	0	3
P12. Pontis 5.0 Bridge Management System	4,395	34	26	0	0	0
P13. Public Health Information Network	2,988	99	32	1	0	5
P14. Publications Management System	2,546	72	21	0	0	1
P15. Puget Sound Enhancements	2,046	13	15	0	1	0
P16. Tactical Control System	5,855	21	23	1	0	1
P17. Tarrant County Integrated Justice Information System	1,763	63	12	0	0	0
P18. X-38 Fault Tolerant System Services	5,455	38	27	1	1	0
H01. CCHIT	2,430	26	29	0	0	0
H02. CM1	514	0	5	0	0	0
H03. InfusionPump	2,956	6	11	1	1	0
H04. Waterloo	11,810	80	48	1	0	1
C01. Amazing Lunch Indicator	3,241	51	22	0	0	3
C02. EU Rent	492	13	11	0	0	0
C03. FDP Expanded Clearinghouse Pilot	464	2	4	0	0	0
C04. Library System	1,779	38	16	0	0	2
C05. Nodes Portal Toolkit	2,239	4	7	0	0	0
C06. Online National Election Voting	3,240	96	24	4	6	1
C07. Restaurant Menu & Ordering System	896	6	7	0	0	0

Chapter 6

Problem of Inconsistency in Textual Requirements Specification

This chapter reflects our publications:

- Šenkýř, D.; Kroha, P. Problem of Inconsistency in Textual Requirements Specification. In: *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, Porto, 2021. [A.4]
- Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies*. SciTePress, Madeira, 2018. [A.10]

In this chapter, we investigate the inconsistency problem in the textual description of functional requirements specifications. In the past, the inconsistency problem was investigated using analysis of the UML models. We argue that some sources of inconsistency can be revealed in the very first steps of textual requirements analysis using linguistic patterns that we developed. We cluster the sentences according to their semantic similarity given by their lexical content and syntactic structure. Our contribution focuses on revealing linguistic contradictions (e.g., a combination of passive voice, antonyms, negated synonyms, etc.) of facts and rules described in different parts of requirements together with contradictions of the internally generated model.

6.1 Problem Statement

Textual requirements do not always contain consistent information about the system to be constructed. The reason is that they might be written by independent groups of stakeholders who have different interests, goals, backgrounds, and/or (incomplete) knowledge of the domain.

The conflicts may erase in the case when more than one requirement, i.e., a set of requirements, refers to the same object of the domain. It will be denoted as an overlap between requirements [110].

In this chapter, we identify two kinds of *the core sources of the inconsistency* of textual requirements. First, semantic overlaps of requirements sentences, e.g., at least two contradicting assertions that concern the same subject and the corresponding verb with the object exist. Second, the incompleteness of requirements discussed in the previous chapter.

Our goal is to reveal at least some of the inconsistencies and generate questions or warning messages that have to be answered by domain experts and analysts. The answers have to be used to edit the text of requirements.

The text of the requirements is the only source of information. We divided the text analysis into two phases. The first phase builds a domain model from sentences using the grammatical inspection (as proposed in Chapter 3) – see Example 6.1 in Section 6.3.2. The second phase provides an analysis of sentences using our linguistic patterns with the goal of finding contradictions in them. Part of it is the using information already stored in the created domain model.

We use the *part-of-speech* and *dependency analysis* of sentences to construct the corresponding oriented graphs, and we analyze sentences that describe the same event. We present our patterns that may indicate candidates for inconsistency.

6.2 Related Work

The problem of inconsistency of requirements specification has been investigated since the late '80s [61]. As basic papers and surveys, we can denote [111] and [71].

Some works are based on semi-formal specifications in the form of UML diagrams [123] and use methods of formal specifications [103] to check the consistency. However, the adoption of formal methods is still not widely accepted by industries [34]. Other works use ontologies [77] or OCL [23].

Our approach uses informal specifications written in natural language. There is some similarity to the approach described in [46]. The difference is that in [46], the authors use only RDF triplets representation of the textual requirements specifications, whereas we use the complete text. The knowledge represented in RDF consists of triplets ⟨subject, predicate, object⟩. However, it reduces the semantic meaning of most sentences drastically because of the omitted information carried by omitted words in sentences. We use a structure similar to RDF triplets only to cluster sentences.

Computational linguistics is engaged in a problem of text entailment (in query answering). The edit distance of two sentences (text and hypothesis) is used as a measure. A similar approach to the contradiction of two sentences is used in [27] – see Section 6.3.1.1. The difference to our approach is that linguists do not use the information stored in the domain (UML) model [104] – see Example 6.1 in Section 6.3.2 to check the completeness that can be a source of inconsistency.

6.3 Sources of Inconsistency

In our investigation, we distinguish the following sources of inconsistency: inconsistency caused by semantic overlaps of sentences (Section 6.3.1), inconsistency caused by incompleteness (Section 6.3.2), and inconsistency caused by external information. We do not discuss the last case here because we analyze it in the next chapter regarding *default consistency rules*.

6.3.1 Semantic Overlaps as Sources of Inconsistency

The idea of *inconsistency* starts from the intuitive concept of contradiction, which means that *a statement* and its *negation* are found to hold simultaneously [46]. A semantic overlap between two sentences occurs if these two sentences express their statements about the same situation. The problem is how to reveal the contradiction of statements because they are often expressed in some "camouflaged" forms using synonyms, antonyms, and other "linguistic tricks" (Section 6.3.1.1).

6.3.1.1 The Linguistic Sources of the Inconsistency

In [27], there is a list of *contradiction types* from a linguistic point of view. We have adapted it for the purposes of our patterns as follows:

- using *antonyms*,
- using a *negation*,

- using a combination of *synonyms* together with *changed roles of subject and object*, *passive voice*, and *negation*, e.g., *"the user can edit a document"* contra *"the user cannot correct a document in this mode"* (here, we suppose that the verbs *"to edit"* and *"to correct"* have the same meaning) contra *"a document cannot be corrected by the user"*,

- using *numerically different data*, e.g., *"you will start the function by double click"* contra *"you will start the function by one click"*,

- using *factive contradiction* in the sense of attributes of the subject,

- using *lexical contradiction*, e.g., *"to obtain results stay joined and wait"* contra *"to obtain results restart the application"* contra *"to obtain results restart the system"*,

- using *world knowledge* to indicate the contradiction, e.g., *"there is public access to your private data"*.

Additionally, some words may change, influence, or limit the sense of the sentence, e.g., *but*, *except*, *however*, *instead of*, *when*, *so that*, *that*.

6.3.2 Inconsistency between the Text and the UML Model

Using the grammatical inspection in the first phase of our process, we construct a skeleton of the domain (UML class) model, e.g., classes, attributes and their values, and methods. When we later compare the sentences in which the attribute and its values are mentioned, we test whether all attribute values are mentioned in formulations defining decisions about the object's behavior. If some of them are not mentioned, then this incompleteness can cause inconsistency because it may happen that the behavior is not defined for the missing attribute values. Further, we test whether the chains of applied methods are the same in sentences of similar semantics – see Example 6.2. In this way, we can find conflicts among descriptions of instances, classes, and relations of the static model and also among descriptions of the sequence diagram and state diagram of the dynamic model. We search for sentences that describe the same event and compare them. More details are given in Section 6.4.1.

In general, we suppose that the textual description of the requirements specification may contain:

- the sentence S_1, in which an attribute value of an object O is expected to be V_1 to start a specific action A in a specific context *Cont*,

- the sentence S_2, in which an attribute value of an object O is expected to be V_2 to start the same specific action A in a specific context *Cont*.

In the following Example 6.1, we show a situation of the constructed domain model concerning the same specific action A, the same object O, and different attributes and their values.

Example 6.1: Attributes of a button.

Maybe the following sentences are a part of a requirements specification:

1. *To exit the application, the user has to press the red button.*
2. *To exit the application, the user has to press the square-shaped button.*
3. *To exit the application, the user has to press the "Exit" button.*

Analyzing these requirements, we can derive a class called `Button`, which has the following attributes: `color`, `shape`, and `label`.

These sentences as part of the requirements are confusing, but they are not inconsistent because a red, square-shaped button with the label "exit" may exist, i.e., it is possible to construct an object of the `Button` class, whose attributes `color`, `shape`, and `label` have the described values.

Our task is to generate a message saying that there is a suspicion of inconsistency. In any case, this is not a preferred style of writing textual requirements.

Another case, we found, concerns the sequence of actions that is stored in a sequence diagram. If the chain of actions differs in different sentences that should have the same effect, we generate a warning message.

Example 6.2: A missing segment in a chain of actions.

Sentence 1: To edit a file, the user has to open the file, make changes, save the file (or save the file as), and close the file.

Sentence 2: To edit a file, the user has to open the file, make changes, and close the file.

6.4 Our Approach

Our motivation and goal are to identify *suspicious textual formulations* that could be the source of inconsistency.

The idea of our approach is shown in Fig. 6.1. The basic structure is given by *subject–verb–object*. However, they can be expressed by related synonyms, antonyms, direct negation, and numerical data. So, the spectrum of possible descriptions of one statement is semantically very rich. Our patterns should reveal it. Practical examples of data used for experiments are given in Section 6.5.1.

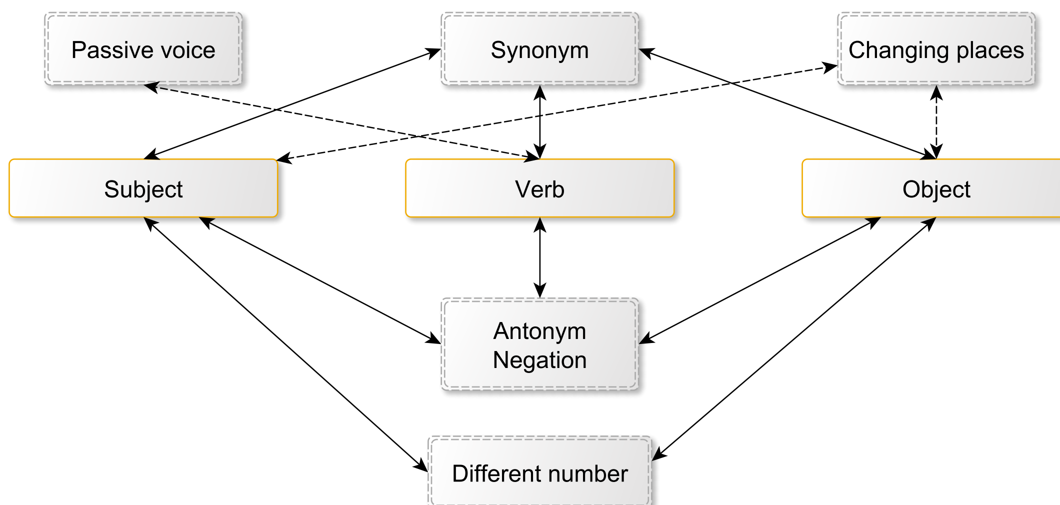


Figure 6.1: The idea of inconsistency patterns.

6.4.1 Model Construction and Semantically Similar Sentences

For our methods, the sentences containing the standard *couple of subject and verb* or the *to-infinitive clauses* are interesting. We benefit from mapping such sentences to *acts*. One reason is that one sentence can represent multiple acts. Our *act* is a tuple containing the original *subject* of the sentence, the original *verb* of the sentence, the original *object* of the sentence, an indicator of whether the sentence contains *negation*, an *auxiliary verb* if present, an *adverbial modifier* (e.g., *automatically*) if present, a *predicate* (e.g., *only one*, *each*) if present, a *condition act* if present, *what* is the point of interest if recognized, *who* is responsible for the action if recognized.

To reduce the complexity of the analysis, we define semantically similar acts as follows:

- C_1: acts having the same subject, verb, and object,
- C_2: acts having the same subject and verb (passive mode),
- C_3: acts having the same subject (passive mode),
- C_4: acts having the same subject and object,
- C_5: acts having the same what-part and verb.

In the following presented *sentence patterns*, we reuse the already introduced graphical representation (as shown in Chapter 3) involving *part-of-speech tags* and *dependency types*. Also, we reuse the shortened notation `NN*`, representing the usual situation when a composition of several nouns represents *subjects* and *objects*.

In the first phase, we need to consider the following steps.

- *Negation recognition.* We check the pure negation of verbs (*can–can not–can't*) or compound nouns via Pattern in Fig. 6.2. In Fig. 6.3, there is this pattern matched as a part of matching standard *subject–verb–object(s)* pattern.

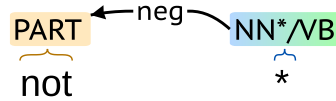


Figure 6.2: Pattern #1 (pure negation).

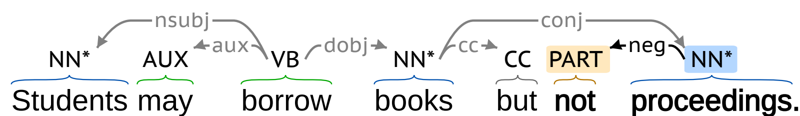


Figure 6.3: Matched pattern #1 (pure negation).

Besides the *not* part, there are other words influencing the negation of meaning. Let us show the pattern of the word *except* in Fig. 6.4 as an example.

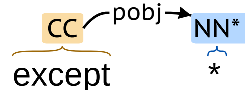


Figure 6.4: Pattern #2 ("except").

- *Coreference recognition.* Clustering of sentences based on subject or object is challenging because of the *coreferences* of pronouns. This means that we are not simply looking for all sentences that have the same subject or object.

- *Predicates recognition.* We identify parts of sentences that describe *predicates* – numeric ones (see Fig. 6.5) or determiner ones (see Fig. 6.6).

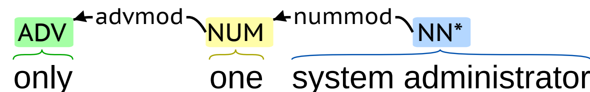


Figure 6.5: Pattern #3 (numeric predicate).

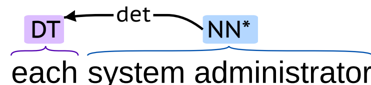


Figure 6.6: Pattern #4 (determiner predicate).

The determiner should be represented by words like *each*, *every*, *all*, *any*, etc. We use the simple first-order logic definition saying that a predicate is a statement about

the properties of an object that may be true or false depending on the values of its variables. Predicates can affect *actors of relations*, as with the relation *"borrow"* in the example sentence in Fig. 6.7 or *aspects and limits of classes* via the auxiliary verb *"be"* (see Fig. 6.8). Mapped predicates also may indicate *cardinalities* of relations between actors or objects. We find predicates concerning the same objects and the same attributes.

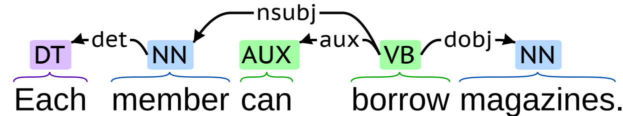


Figure 6.7: Relation actor predicate.

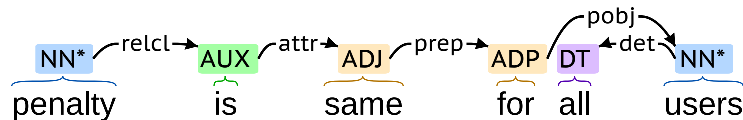


Figure 6.8: Pattern #5 (predicate and auxiliary verb).

- *Condition recognition.* Describing actions often includes a conditional part. The conditional parts of two sentences may have the same *subject–verb–object*. If one sentence considers the positive case and another sentence considers the negative case, then there would be a conflict generating a warning because of the negation of the action. We use these conditional parts as separate conditional acts (affecting standard acts) in the conflict resolution of the standard acts.

6.4.2 Example – Library Information System

To discuss our approach, we use and extend the simple example of *Library information system requirements* taken from [110].

Example 6.3: Library information system: functional requirements.

FR 1. *Users of the Library information system are students and staff members.*

FR 2. *Users borrow books.*

FR 3. *Holding an item is limited in time.*

FR 4. *Holding an item beyond the time limit carries a penalty, which is the same for all users.*

FR 5. *Students may borrow books but not proceedings.*

FR 6. *Staff members can borrow both books and proceedings.*

FR 7. *Students may borrow an item for up to 10 days. Holding an item beyond this period carries a penalty of 50 p per day.*

FR 8. *Staff members can borrow an item for up to 30 days. Holding an item beyond this period carries a penalty of 10 p per day.*

FR 9. *Each user except an administrator needs to change his/her password every three months.*

The purpose of this example is to show the contradiction caused by *teaching students*. To reduce the ambiguity of the previous intentionally worded sentence, we state that we mean students who study and teach at the same time. We use this situation to illustrate our approach. When using an algorithm to analyze these textual requirements instead of a human being analyst (like in [110]), we can see the following problems.

1. In **FR 1**, there are two missing rules. In the next chapter, we call them *default consistency rules*. The first default consistency rule is *"All information systems need an administrator, who is a unique, specific user"*. The second default consistency rule is *"All libraries need library staff members, who are specific users of the library information systems, too"*.

So, the set of users is incomplete, and it has to be completed. In this case, we model the classes and the *is-a* relationship between a *user* (as a superclass) and subclasses *student* and *university-staff-member*. Then, we generate a question asking about the completeness of this relation, i.e., whether the superclass *user* has only these two mentioned subclasses. The right answer contains not only the missing administrator and the missing library staff member but also the missing teaching students. After the subclasses *administrator*, *library staff member*, and *teaching student*[1] are included in the requirements, our tool TEMOS will complain of incompleteness because the loan time periods and penalties are not defined for these three subclasses.

2. In **FR 2**, the formulation *"Users borrow books"* is misleading because users not only borrow books. They hold them and have to return them, too. Without this information, the time limit mentioned in FR 4 and the period mentioned in FR 7 and FR 8 make no sense. Using linguistic analysis, we find that there are only the verbs *"borrow"* and *"change (the password)"*. Here, we can see the incompleteness as the source of the inconsistency again.

3. In **FR 3**, this formulation is understandable to a human being but not to an algorithm. *"Holding an item"* means *"holding a borrowed item"*. Also, it should be stated that the item represents a book and a proceeding in this context. The domain expert should improve the formulation: *"Users borrow books or proceedings, hold them, and return them."*

4. In **FR 5**, the *pattern #1 (pure negation)* matches the statement as shown in Fig. 6.3.

[1] It would be even better to design this using roles.

5. In **FR 7** and **FR 8**, we can detect an inconsistency with **FR 4**. Either all library users pay the same penalty, or a different penalty is applied according to the library user status.

6. In **FR 9**, there is *pattern #4 (determiner predicate)* matched twice (*"each user"* and *"every three months"*) and *pattern #2 (except)* is matched once.

6.5 Experiments and Results

In this section, we first present the data used in our inconsistency checks. Second, we present an evaluation of our proposed methods using the entire dataset (Section 3.6.1) in the form of the frequency of detected issues.

6.5.1 Data

We decided to focus on antonyms; negations; a combination of synonyms together with swapped roles of subject and object, passive voice, and negation; and numerically different data described in Section 6.3.1.1.

6.5.1.1 The List of Antonyms

The list of antonyms used to test the inconsistency: *source—destination*, *first—last*, *all—selected*, *open—close*, *at the front—at the end*, *send—receive*, *numeric—alphabetic*, *high—low*, *valid—invalid*, *insert—delete*, *locked—unlocked*, *unique—duplicated*, *chronologically sorted—alphabetically sorted*, *able—unable*, *in only one—in one or more*, *private—public*, *at the bottom—at the top*, *expanded list—reduced list*, *upper-right corner of the window—upper-left corner of the window*, *undo command—redo command*, *enable—disable*, *more—less*, *appear—disappear*, *manually—automatically*.

6.5.1.2 The List of Negations

The list of negations used to test the inconsistency:

- simple negations of verb forms (modal verbs, e.g., *"it is the same"*, *"it is not the same"*, standard verbs, e.g., *"it exists"*, *"it does not exist"*),
- negation of the similar meaning, e.g., *"imported data can't be modified"* contra *"you can modify the imported data"*,
- complex negations of the sentence meaning, e.g., *"you can display it if you are in the review mode"* contra *"you can display it at any time"*.

6.5.1.3 The List of Numerically Different Data

The numerical data can be given by digits, by words, or by other means, e.g., Max-Int. We test similar sentences and generate warnings when some suspicious formulations are found. For example: *"There is an unused field that should be set to zero."* contra *"There is an unused field that should be set to Max-Int."*

The numerically different data also includes statements about object uniqueness, as shown in the following example.

Example 6.4: A unique object.

Sentence 1: *The system has only one system administrator.*
Sentence 2: *All system administrators have the same access rights, but each of them has his/her password.*

Here, *pattern #3 (numeric predicate)* is matched as shown in Fig. 6.9. We generate a warning message because *"only one system administrator"* in the first sentence and *"all"* and *"each of them"* (system administrators) in the second sentence contradict each other.

Pattern #4 (determiner predicate) is matched two times. First, it is applied in the part *"All system administrators"*. Second, it is applied in the part *"each of them"* when the co-reference to system administrators is resolved.

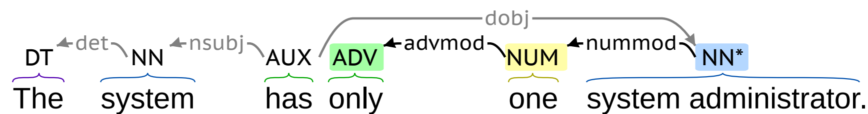


Figure 6.9: Consistency rule from Example 6.2.

6.5.1.4 The Incomplete List of Items

We identify this source of inconsistency using the domain (UML) model that we obtained in the first phase of the processing. In the model, the attribute values are given by a list of values, but in some sentences, only a subset of them is mentioned. Often, this is a signal that a reaction to some state of the system has been forgotten. In this case, the incompleteness (as described in the previous chapter) is the source of the inconsistency.

6.5.2 Results

Table 6.1 summarizes the results of experiments of our presented inconsistency detection approach using the data set introduced in Section 3.6.1. The results include a breakdown according to individual clusters defined in Section 6.4.1, where we present the count of each

cluster for each case of the data set, the cardinality of each cluster (the column *"card."*), and the number of detected issues in each cluster (the column *"is."*).

Table 6.1: Evaluation of methods detecting inconsistency.

	C1			C2			C3			C4			C5		
Case	count	card.	is.	count	card.	is.	count	card.	is.	count	card.	is.	count	card.	is.
g02-federalspending	4	2–3	0	0	0	0	0	0	0	10	2–7	0	2	2–3	0
g03-loudoun	4	2–2	0	0	0	0	0	0	0	11	2–4	0	5	2–2	0
g04-recycling	1	3–3	0	0	0	0	0	0	0	4	2–3	0	1	3–3	0
g05-openspending	1	2–2	0	0	0	0	0	0	0	3	2–2	0	1	2–2	0
g08-frictionless	6	2–12	0	0	0	0	0	0	0	5	3–14	0	2	2–11	0
g10-scrumalliance	11	2–4	0	0	0	0	1	2–2	0	20	2–7	0	9	2–4	0
g11-nsf	4	2–2	0	1	2–2	0	0	0	0	7	2–2	0	4	2–2	0
g12-camperplus	2	2–2	0	0	0	0	0	0	0	2	2–2	0	1	2–2	0
g13-planningpoker	5	2–3	0	0	0	0	0	0	0	9	2–8	0	4	2–3	0
g14-datahub	5	2–3	0	0	0	0	0	0	0	6	2–19	0	6	2–3	0
g16-mis	1	3–3	0	0	0	0	0	0	0	4	2–3	0	1	3–3	0
g17-cask	12	2–6	1	0	0	0	1	2–2	0	15	2–9	2	12	2–4	0
g18-neurohub	7	2–18	0	0	0	0	0	0	0	12	2–18	0	7	2–18	0
g19-alfred	16	2–24	0	0	0	0	0	0	0	20	2–35	0	13	2–24	0
g21-badcamp	8	2–6	0	1	2–2	0	0	0	0	12	2–8	0	7	2–6	0
g22-rdadmp	6	2–6	0	6	2–4	0	2	2–2	0	14	2–6	0	13	2–6	0
g23-archivesspace	6	2–4	0	1	2–2	0	0	0	0	8	2–5	0	6	2–2	0
g24-unibath	7	2–3	1	0	0	0	0	0	0	10	2–7	1	8	2–2	0
g25-duraspace	7	2–4	0	0	0	0	1	2–2	0	12	2–7	0	6	2–4	0
g26-racdam	4	2–5	0	0	0	0	1	2–2	0	18	2–11	0	4	2–5	0
g27-culrepo	7	2–3	0	2	2–2	1	1	3–3	0	16	2–6	0	9	2–3	0
g28-zooniverse	6	2–3	0	0	0	0	0	0	0	7	2–4	0	6	2–3	0
P01. Blit	3	2–2	0	0	0	0	0	0	0	2	2–4	0	0	0	0
P02. CS179G...	1	2–2	0	0	0	0	0	0	0	1	2–2	0	0	0	0
P03. eProcurement	6	2–2	0	5	2–2	0	1	2–2	0	8	2–2	0	8	2–4	0
P04. Grid 3D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P05. Home 1.3	2	2–2	0	0	0	0	0	0	0	2	2–2	0	3	2–3	0
P06. Integrated...	7	2–4	0	0	0	0	1	2–2	0	7	2–7	0	0	0	0

Case	C1			C2			C3			C4			C5		
	count	card.	is.	count	card.	is.	count	card.	is.	count	card.	is.	count	card.	is.
P07. Inventory	39	2–20	2	8	2–11	1	3	3–6	0	40	2–20	2	23	2–11	0
P08. KeePass...	1	2–2	0	2	2–3	0	2	2–3	0	1	2–2	0	3	2–3	0
P09. Mashbot	1	2–2	0	0	0	0	1	2–2	0	1	3–3	0	0	0	0
P10. MultiMahjong	8	2–5	1	0	0	0	1	2–2	0	9	2–6	1	1	2–2	0
P11. Nenios	1	2–2	1	0	0	0	1	2–2	0	4	2–2	1	1	3–3	0
P12. Pontis 5.0...	10	2–6	0	4	2–4	0	3	2–2	0	10	2–6	0	5	2–4	0
P13. Public...	8	2–9	0	2	2–4	0	3	2–2	0	9	2–9	0	7	2–8	0
P14. Publications...	15	2–5	0	1	2–2	0	0	0	0	17	2–5	0	4	2–3	0
P15. Puget...	4	2–2	0	2	2–2	0	1	2–2	0	5	2–4	0	2	2–2	0
P16. Tactical...	13	2–33	0	0	0	0	2	2–2	0	12	2–47	0	2	2–3	0
P17. Tarrant...	7	2–8	0	1	2–2	0	1	2–2	0	11	2–12	0	1	3–3	0
P18. X-38 Fault...	35	2–8	1	11	2–8	2	3	2–3	0	40	2–8	1	17	2–8	0
H01. CCHIT	10	2–7	1	1	2–2	0	4	2–2	0	20	2–9	1	2	2–2	0
H02. CM1	0	0	0	0	0	0	1	2–2	0	3	2–2	0	0	0	0
H03. InfusionPump	11	2–5	1	2	3–4	0	4	2–2	0	13	2–5	1	6	2–4	0
H04. Waterloo	92	2–131	1	3	2–3	0	12	2–7	1	84	2–164	3	16	2–16	0
C01. Amazing...	7	2–7	0	4	2–16	0	3	2–4	0	17	2–7	0	12	2–17	0
C02. EU Rent	2	2–4	0	0	0	0	0	0	0	2	2–4	0	0	0	0
C03. FDP...	0	0	0	1	3–3	0	1	3–3	0	0	0	0	1	3–3	0
C04. Library System	8	2–6	0	5	2–3	0	3	2–10	0	9	2–6	0	11	2–3	0
C05. Nodes...	18	2–6	0	0	0	0	0	0	0	28	2–19	0	0	0	0
C06. Online...	17	2–7	0	1	2–2	0	2	2–2	0	23	2–7	0	3	2–4	0
C07. Restaurant...	3	2–4	0	0	0	0	0	0	0	4	2–4	0	1	2–2	0

Legend: **C1, C2, C3, C4, C5** – clusters defined in Section 6.4.1, **card.** – the cardinality of each cluster, **is.** –the number of detected issues in each cluster

Chapter 7

Problem of Default Consistency Rules in Textual Requirements Specification

This chapter reflects our publications:

- Šenkýř, D.; Kroha, P. Problem of Inconsistency and Default Consistency Rules. In: *New Trends in Intelligent Software Methodologies, Tools and Techniques*. IOS Press, Amsterdam, 2021. [A.3]
- Šenkýř, D.; Kroha, P. Problem of Semantic Enrichment of Sentences Used in Textual Requirements Specification. In: *Advanced Information Systems Engineering Workshops*. Springer, Cham, 2021. [A.5]

Incompleteness and inconsistency are well-known defects of textual requirements. Some missing parts are detectable as we proposed in the previous chapters, e.g., when the state of the object is, by definition, described using four different values, but the connected use case scenarios are prepared only for three values out of four.

However, some missing rules and definitions are not detectable just based on the provided text because the indicators are missing in the text. In some situations, the authors of the requirements forgot them by mistake. Nevertheless, in some situations, they do not mention them intentionally because they think that the rules and definitions are so obvious. This may not be true for teams with different problem domain knowledge. We dedicate this chapter to such types of missing parts, and we present our approach using the existing web application.

7.1 Problem Statement

In this chapter, we focus on such facts and rules that are so obvious to domain experts that they do not even mention them to the analysts during the discussions about the product to be constructed. However, what is very obvious to stakeholders may not be obvious to analysts. We call such rules *default consistency rules*. The problem is that they are not described in the textual description of requirements, i.e., we cannot simply use the analysis concerning *inconsistency* as we have done it in the previous chapter.

The missing description of the default consistency rules represents incompleteness of the requirements, and it causes inconsistency with all unpleasant consequences. In this section, we describe our approach to the problem of how the missing information can be identified in requirements and how it can be found (sometimes) in external sources. We show a motivational example and explain our method.

Our solution is based on the assumption that the missing default consistency rules can be found somewhere in external sources of information, e.g., on the Web, and then linked to the text of requirements to enrich them. After this replenishment phase is done, we analyze the completed text of requirements using our methods that we developed to identify inconsistency of textual requirements, as proposed in Chapter 6.

This problem is complex because we cannot always find such rules in external sources explicitly. We need to provide text mining of web pages that concern the same semantic context as the expected system's functionality.

7.2 Case Study – Part 1: Missing Consistency Rules in Chords Generation

To introduce the default consistency rules concept and to prove its existence, we show a practical example taken from an existing application.

We found a web application [21] that generates chords for 4-string instruments. We use it to illustrate the problem to be solved. We do not have access to its textual requirements

Figure 7.1: Example of a not playable chord. [21]

specification, but we can see the results, i.e., we can see that one of the default consistency rules was not included in the requirements. In Fig. 7.1, it should be shown how to play the chord Ebm^9 on a 4-string banjo with tuning C-G-D-A.

The problem is that the fingering shown in Fig. 7.1 is not realistic because no musician has such a wide finger span to play tones in position three and position ten simultaneously. The authors of the application have seen it and solved this mistake by a notice (see the arrow in the figure): *"All chords are calculated and therefore may not be playable!"*. However, users usually look for playable chords.

So, the *default consistency rule* – *"All chords have to be playable"* – is missing, i.e., it has been forgotten because it is obvious to domain experts.

7.2.1 Chord Generation Requirements

Suppose we have to specify a program generating chords fingering for 4-string music instruments. Besides the rules of the chord construction, we have to include the default consistency rule: *"The generated chords have to be playable. The playability of a chord means that all chord tones come from different strings simultaneously (Fig. 7.2), and the finger span over positions is limited to 4."*

Now, we suppose to know the missing rules, and we explain how to find them semi-automatically. We list the basic properties of playable chords and denote them as missing default consistency rules.

- **Default consistency rule No. 1**:

  The chord tones sound simultaneously (we abstract from arpeggio at this moment), i.e., every tone of a chord has to be played on a unique string if they should sound simultaneously. This means that we have to omit all automatically generated solutions in which there is more than one tone on one string (see Fig. 7.2).

- **Default consistency rule No. 2**:

  The chord fingerings on the fingerboard respect the limited hand span of musicians, i.e., the chord finger positions on the fingerboard have to be in a specific frame of finger span, e.g., 1–4, because of the realistic finger span of musicians (see Fig. 7.1).

In the following parts of this chapter, we introduce our method that consists of searching for additional information in external sources to enrich the requirements.

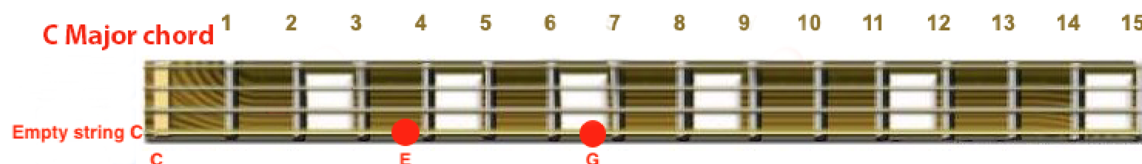


Figure 7.2: Example of a not playable chord – three tones on one string.

7.2.2 Our Approach – Using External Context to Identify the Missing Default Consistency Rules

Our method consists of collecting and importing such sentences from web sources (e.g., Wikidata, Wordnik, etc.) that have a semantic similarity with the topic described in the textual requirements. We cluster these sentences and analyze them to find some semantic enrichment that could be used to reduce the incompleteness of the requirements.

7.2.3 Construction of Pseudo-Questions

We define pseudo-questions as patterns that are partially filled in by:

- words, their types, and their syntactic roles in a sentence, e.g., ⟨***chord***, **noun**, **SUBJECT**⟩,
- empty containers that match any part of a sentence based on the restrictions.

These are definitions of pseudo-questions used when revealing semantic enrichment regarding a concrete noun N:

- **Pseudo-question Type 1**:
  optional ⟨any **adjective**⟩
  ⟨N, **noun**, **SUBJECT**⟩
  ⟨***is***, **verb**, **PREDICATE**⟩
  ⟨empty container, **noun**, **OBJECT**⟩

- **Pseudo-question Type 2**:
  ⟨empty container, **noun**, **SUBJECT**⟩
  ⟨***is called***, **verb**, **PREDICATE**⟩
  optional ⟨any **adjective**⟩
  ⟨N, **noun**, **OBJECT**⟩

- **Pseudo-question Type 3**:
  ⟨N, **noun**, **SUBJECT**⟩
  ⟨empty container, **verb**, **PREDICATE**⟩
  ⟨empty container, **noun**, **OBJECT**⟩

7.2.4 Semantic Similarity of Sentences

The evaluation of semantic similarity between two sentences belongs to the most important tasks in natural language processing and the derived topics like information retrieval, text mining, and query-answer systems.

Generally, the semantic similarity of two sentences determines how similar the meaning of two sentences is. In paper [114], there are given some definitions of semantic similarity of sentences (semantic similarity between words of the sentences like cosine similarity, and syntactic measures like path-based approach or feature-based approach).

We need to measure the semantic similarity of sentences in the third step of our Algorithm 7.1 (Section 7.2.6). We use only a very simple method based on RDF, i.e., in the external sources, we search for sentences containing the same triplet ⟨**subject**, **predicate**, **object**⟩ or their subsets as our pseudo-question. However, the process is not simple because of synonyms, antonyms, negations, co-reference, and other linguistic challenges that influence sentence meaning. Each pseudo-question is a seed of a cluster, to which we cluster the found semantically similar sentences.

7.2.5 Semantic Enrichment of Sentences

To find the semantic enrichment of sentences, we compare each pseudo-question with sentences in its cluster. The goal is to find enrichment but exclude a sentence with the same meaning. We have proposed our measure of semantic enrichment based on graph representation in [A.5]. It contains its lexical part (node names) and its structural part (roles of nodes and edges representing sentence structure). We use the graph representation as node-node matrices and denote the matrix of the pseudo-question as G_0 and the matrices of sentences in a cluster as G_1, G_2, ... Then we investigate whether the graph G_0 is a

subgraph of graphs G_1, G_2, ... of sentences in the corresponding cluster. If we find such cases, we hold the differences $G_1 - G_0$, $G_2 - G_0$, ... for semantic enrichments of G_0, i.e., semantic enrichments of the pseudo-question.

Based on these semantic enrichments (additional nodes and edges), we generate a question asking a human analyst and domain expert to which extent these enrichments should be included in the next version of the requirements.

7.2.6 The Process of Semantic Enrichment of Sentences

The whole semantic enrichment process runs iteratively, and it includes the steps illustrated in the following algorithm.

Algorithm 7.1: The semantic enrichment of sentences.

1. information extraction from sentences in the original requirements
2. searching and clustering similar sentences from external sources
3. information extraction from the found similar sentences
4. identification and extraction of semantic enrichment from the found sentences
5. linguistic inference (replaced by a human intervention here)

Once the analyst completes the enrichment phase, we can analyze the completed text using our inconsistency patterns. We discussed the problems of requirements inconsistency in the previous chapter, and we developed some patterns that indicate inconsistency of textual requirements. It may be used for the next iteration.

Below, we describe each step of Algorithm 7.1.

7.2.6.1 Step 1: Information Extraction from Original Requirements

The first goal is to the identify entity and attribute candidates represented by nouns. We describe the corresponding method in Chapter 3. We eliminate all such nouns that have too general meaning, e.g., *application*. Such nouns are used in many projects, and they do not carry the semantic information typical for the specific project.

7.2.6.2 Step 2: Searching and Clustering Similar Sentences

We generate pseudo-questions (Section 7.2.3) from selected nouns from the previous step. Our pseudo-questions are patterns, not questions in the sense of linguistics. The goal of this step is to find sentences in external information sources that have semantic similarities with the generated pseudo-questions. To complete the missing knowledge, we use the following resources (hereinafter referred to as *proposed information sources*):

- knowledge bases, e.g., Wikidata, including dictionaries, e.g., Wordnik [129],
- a Web search engine, e.g., Google,
- internal company documents with definitions, if available.

Thus, obtained sentences are then clustered with the extracted part from the first step, depending on the type of pseudo-question.

When this level of clustering is done, we introduce the second level of clustering using subject and object, both including optional adjectives. This clustering is beneficial because two different sentences should have the same semantic meaning. For example, let us use sentences matching pseudo-question type 1 and type 2.

- *"A chord is a harmonic set of tones."*
- *"A harmonic set of tones is called a chord."*

In the end, both sentences contain the same semantic information. The second level of clustering eliminates such duplicates.

Of course, we need to omit the duplicates on both levels of clustering. There is a chance of clustering similar sentences with the same semantic meaning, such as already included sentence extracted from the original requirements.

7.2.6.3 Step 3: Information Extraction from Similar Sentences

On the level of clusters created in the previous step, we find the important differences in the added sentences found by the pseudo-questions. We expect to find a semantic enrichment (Section 7.2.5) of pseudo-questions, e.g., different definitions or some additional properties of entities presented as adjectives that are used in external sources but not in the requirements to the same nouns.

7.2.6.4 Step 4: Enrichment

In this step, we propose the insertion of the missing information, i.e., the information of semantic enrichment, into the textual requirements. The hints that could potentially enrich the text of requirements are given to a human expert – it is then their task to complete the text of requirements.

The number of external information sources used for the semantic enrichment is one of the factors of scalability. As the number of resources grows, the number of possibly generated hints increases. Therefore, we sort the collection of hints. First, we consider hints where the same semantic information came from multiple sources and was clustered into a single hint. Then, we prioritize hints that came from knowledge bases, dictionaries, or internal documents (if available) over hints from a plain Web search.

7.2.6.5 Step 5: Human Intervention

The result of the last steps is a generation of questions on domain experts and analysts, who decide how to fill in the information gaps in the requirements.

7.3 Case Study – Part 2: Applying Our Approach

The missing default consistency rule: *"The chords have to be playable"* appears in external information sources clearly. For example, the playability of chords is mentioned in [95] as a property of chords: *". . . but the number of playable chords is very large and depends on the players."* There is the adjective *playable* used together with the noun *chord*.

As described above, we split this default consistency rule into two default consistency rules on a more detailed level.

7.3.1 The Missing Consistency Rule No. 1

In this section, we show how to find the missing default consistency rule No. 1: *"The chord tones sound simultaneously."* It means the generated tones cannot be played on the same string. We start by searching for nouns in descriptions of the project defined in Section 7.2.1, because they have the main impact on semantics.

In the very brief description (Specification 1), we find only nouns: *application*, *chord*, and *4-string music instrument*. From the linguistic point of view, the words *4-string music instrument* is not a single noun, but it represents the concept of biwords from information retrieval, such as *New York*. It means that these three words (*4-string music instrument*) are used to denote one object.

In the brief description of the project goal (Specification 2) in Section 7.2.1, we find nouns: *application*, *definition*, *chord*, *type*, *scale*, *user interface*, *tuning*, *4-string music instrument*, *chord diagram*, *finger position*, *fingerboard*.

The nouns *application*, *definition*, *type*, *scale*, *user interface*, and *program* are contained in all requirements specifications, so they can be eliminated: they are not typical for the context-specific semantics, i.e., they do not contribute to the unique semantics of the given project. Thus, we skip these nouns.

We suppose that the nouns *chord*, *tuning*, *4-string instrument*, *chord diagram*, *finger position*, and *fingerboard* build the set of words that can be used to find the context of our project in external sources.

Now, we illustrate the steps of our method that we described in Algorithm 7.1 (Section 7.2.2).

7.3.1.1 Step 1: Information Extraction from Original Requirements

As described, we identify all entity and attribute candidate nouns in the textual description of requirements, and we skip the general ones. This way, we got these nouns: *chord*, *tuning*, *string instrument*, *chord diagrams*, *finger position*, and *fingerboard*.

7.3.1.2 Step 2: Searching and Clustering Similar Sentences

We use our specific patterns, called **pseudo-questions** (Section 7.2.3), that are constructed to find a definition. When we consider the noun *chord*, it looks like this:

- **Pseudo-question Type 1**:
  optional ⟨any **adjective**⟩
  ⟨***chord***, **noun**, **SUBJECT**⟩
  ⟨***is***, **verb**, **PREDICATE**⟩
  ⟨empty container, **noun**, **OBJECT**⟩
  - Example of typically matched sentence: *"A* ⟨optional adjective⟩ *chord is a* ⟨any rest of the sentence including object⟩."
- **Pseudo-question Type 2**:
  ⟨empty container, **noun**, **SUBJECT**⟩
  ⟨***is called***, **verb**, **PREDICATE**⟩
  optional ⟨any **adjective**⟩
  ⟨***chord***, **noun**, **OBJECT**⟩
  - Example of typically matched sentence: "⟨any begin of the sentence including subject⟩ *is called a* ⟨optional adjective⟩ *chord."*
- **Pseudo-question Type 3**:
  ⟨**chord**, **noun**, **SUBJECT**⟩
  ⟨empty container, **verb**, **PREDICATE**⟩
  ⟨empty container, **noun**, **OBJECT**⟩
  - Example of typically matched sentence:
    "A chord ⟨any predicate⟩ ⟨any rest of the sentence including object⟩."

7.3.1.3 Step 3: Information Extraction from Similar Sentences

To find the missing information, we assume that such information exists in one or more *proposed information sources*. Querying the sources, we try to find a definition for each noun on the list, e.g., for the noun *chord*:

- *"A chord is a combination of three or more tones sounded* ***simultaneously***" [20],
- *"When three or more notes are* ***sounded together****, the combination is called a chord*" [20],
- *"A chord, in music, is any harmonic set of usually three or more notes (also called "pitches") that is heard as if sounding* ***simultaneously***" [20].

7.3.1.4 Step 4: Enrichment

Hereafter, we compare our artificial definition and the definition(s) found. Then, we convert the difference into a question. The generated questions (for our case study):

1. The difference between the chord concept found in the requirements and the chord concept found in proposed information sources is the property described by:
   a) *"a chord is a harmonic set of notes"* ⟶ what does mean *harmonic*?

b) *"tones sounded simultaneously"* or *"notes sounded together"* or *"notes often sounded together"*
$\longrightarrow$ *tones* equals *notes*?
$\longrightarrow$ *simultaneously* equals *together*?
$\longrightarrow$ *chord* means *together*?
Here, we can reuse our approach of synonyms recognition, as presented in Chapter 4.

2. Is there a rule that has to be satisfied due to the chord has this property?
3. Should this rule be included into the set of the consistency rules of the requirements?

7.3.1.5 Step 5: Human Intervention

The answers formulated by a domain expert (exemplified, again, for our case study):

1. *"If you want to play a chord as one sound, then each tone of the chord has to be played on a unique string."*
2. *"If you want to play a broken chord (or arpeggio), then each tone of the chord may or may not to be played on a unique string."*
3. *"The user has to indicate whether he/she will obtain only one-sound chords, i.e., chords whose tones are played simultaneously, or broken chords, i.e., chords whose tones are played in some sequence after each other), or both."*
4. *"Yes, this rule has to be included in the textual requirements."*

One of the missing default rules has been revealed. It concerns the relation between the number of tones of a chord and the number of the participating strings.

7.3.2 The Missing Consistency Rule No. 2

The missing default consistency rule No. 2 (this rule is broken in our case study of application [21]): *"The chord fingering on the fingerboard has to respect the limited hand span of musicians."* It means the chord finger positions on the fingerboard have to be in a specific frame of finger span.

The playability of chords also assumes that the player can put his/her fingers in the prescribed positions on the fingerboard. On the web page [94], we can read: *"For chords that require a wider finger span, ..."* Such sentence corresponds to our *pseudo-question type 3*, and it can be resolved via Algorithm 7.1, similarly to the missing consistency rule No. 1. We found experimentally that for this type of missing consistency rule, it is beneficial to repeat Algorithm 7.1 with the following changes.

- Step 1. When identifying nouns, we remember the corresponding sentence for each noun. Let S be a set of sentences. Then, all nouns of the i-th sentence create a set NS_i.

- Step 2. We create all unique pairs from each set NS_i. These pairs are the subject of the following query phase. The reason to do it in the sentence scope is the semantic connection between concrete nouns in the pair. Since – in this variant of our algorithm – we query pairs, we omit knowledge bases and dictionaries that are constructed for single-term querying. This step generates texts for applying the same patterns from the third step for each noun (please note that the pair of nouns is used just to retrieve the extending texts to process).

Following the found sentence: *"For chords that require a wider finger span, ..."* and the fourth step of Algorithm 7.1, the generated question can be: *"Is there a rule that has to be satisfied due to the statement that* ***chord*** $\longrightarrow$ *relation* ***require*** $\longrightarrow$ ***wider finger span****?"*

The answer formulated by a human specialist should be: *"Considering a 19-fret 4-string banjo and supposing that the fretting range (inclusive of the fingered notes) for an average hand is 4 frets in the first 8 frets, 5 frets in the 9–14 fret region and 6 frets on the remaining of the fingerboard. There may be other rules for other 4-string instruments."*

7.4 Experiments and Results

Following the example of our case study, we present the concrete data processed in the implementation of our solution. We note that using information from external sources

Figure 7.3: Chord definition from WordNet – the scope of the definition exceeds string instruments.

opens the problem of information veracity. In this chapter, we do not solve it. However, to illustrate that such a problem also exists in respected *WordNet*, see Fig. 7.3. It is not true that *"to chord means to play chords on a stringed instrument"* because it is possible to play chords on an organ or an accordion that are not string instruments.

7.4.1 Revealing Consistency Rule No. 1

To reveal the consistency rule No. 1, we first process the following results from the knowledge bases. The data are presented only for the key noun *chord*.

- Wikidata[1]: *A chord is harmonic set of three or more notes.*
- Wordnik [129]: *A chord is a combination of three or more pitches sounded simultaneously.*
- DBPedia[2]: *In music, a guitar chord is a set of notes played on a guitar.*
- BabelNet [86]: *In computing, Chord is a protocol and algorithm for a peer-to-peer distributed hash table.*

One can see that only one definition (BabelNet) is not usable in our case because it targets a different field.

The condition of automatic querying requires a search engine with a public API. For this purpose, we use our configured instance of the Programmable Search Engine provided by Google[3] with the following set-up:

- language: English,
- search the entire web: activated.

The current limitation of the free version is 10,000 requests per day. The response of a query is in JSON format, and it consists of 10 results. Each result provides metadata, including the web page link. We use these 10 result web pages from querying the noun *chord* to create the following statistic in the first numeric column in Table 7.1. It reflects the sentences containing the noun *chord* and the corresponding clusters. The last point of the ratio of reasonable questions to all questions is our subjective categorization. We discuss the ratio in Section 7.4.3.

[1] `https://www.wikidata.org`
[2] `https://www.dbpedia.org`
[3] `https:/programmablesearchengine.google.com`

Resolving consistency rule	**No. 1**	**No. 2**
Search word(s)	*chord*	*chord + fingering*
Sentences containing the noun *chord*	80	18
Sentences matching our pseudo-questions	30	5
Clusters	21	5
— Clusters of cardinality 1	19	5
— Clusters of cardinality 2	2	0
Sentences already used in requirements	2	0
Reasonable questions (hints)	6/19	2/5

Table 7.1: Google Web search: *chord* and *chord + fingering*.

7.4.2 Revealing Consistency Rule No. 2

When revealing the consistency rule No. 2, we were not successful with querying knowledge bases and searching via Google API by a single noun. However, we were successful using the extended version of Algorithm 7.1 (Section 7.3.2) with the pair of nouns *chord* and *fingering*. One of the results was the web page [94] that provides the hint with a *"wide finger span"*.

A statistic similar to the previous consistency rule regarding the noun *chord*, now with the pair of nouns *chord* and *fingering* used as a query, is shown in the second numeric column in Table 7.1.

7.4.3 Discussion

Regarding the ratio of reasonable questions, we recall that questions (hints) are sorted based on our relevance criteria defined in Section 7.2.6.4. Based on the number of information sources, the number of questions (hints) should increase. Therefore, we expect the analysts to consider the top hints from the sorted list primarily.

We also note that due to the nature of online resources, results are variable over time. This primarily applies to a web search. Definitions from knowledge bases and dictionaries, on the other hand, are expected to be more or less stable.

Regarding the scalability, we already mentioned that the amount of questions (hints) is dependent on the number of information sources. In our experiments, we select the first result (definition) of a query from knowledge bases. Although it is possible to include other results without a limit on the first one, we note (based on our experiments) that removing the limit brings rather a disadvantage in the form of definitions from areas unrelated to the main topic of requirements. In any case, another open problem is the selection of definitions based on the main topic of requirements.

Chapter 8

Quality Measurement

This chapter reflects our publication:

- Šenkýř, D.; Kroha, P. Quality Measurement of Functional Requirements. In: *Proceedings of the 18th International Conference on Software Technologies*. SciTePress, Porto, 2023. [A.1]

There are two viewpoints on the quality of a software product. It has to satisfy the requirements specification, and it has to state and imply the needs of all stakeholders. The problem is that the requirements specification usually does not reflect the needs of the stakeholders completely. The reason is that many stakeholders cannot articulate the requirements or do not even know what they want. The analysts help them, but they often do not deeply understand the semantics of the branch (e.g., molecular biology). So, mistakes are practically guaranteed.

For these reasons, it is worth investing effort into formulations of textual requirements before the analysis starts.

8.1 Problem Statement

The development of a software product should follow a software development process model given by the used development methodology. It is a set of related activities that have to be carried out during the production of the software product. As everything goes according to plan, the software product will be completed, tested, and delivered to the customer. At the same time, documentation relating to this product will be created. The problem is that this is not often the case. Due to competition, managers force programmers to hurry at the expense of product quality and documentation quality.

While the quality of the product is tested at the last minute for the customer, the quality of the documentation remains often marked by haste during development because it stays in the software house and because managers mean that the "polishing" of documentation is not as important as the product delivery. There is not always enough time for the documentation to be further improved. As a result, textual requirements specification is often outdated and does not match the latest version of the product completely. This could cause problems with maintenance and further product development.

Customers obtain the program of the software product and the user guide. Users learning from the user guide provide feedback when using the program. We suppose that the user guide is the only updated document (except for the executable program).

Textual formulated requirements specifications are necessary as a base of communication between the customer, the user, the domain expert, and the analyst. Unfortunately, any text suffers from ambiguity, incompleteness, and inconsistency.

The textual description of functional requirements specification has one advantage and many disadvantages.

- Advantage – it is understandable.
  - The analyst can discuss its content with the customer easier in comparison to specific modeling language.
  - In the event of a lawsuit between the company and the customer, the judge has a facilitated role because he understands the assignment and can assess whether the product meets the assignment.

- Disadvantage – it is not precise like all texts. It is easily ambiguous, incomplete, and inconsistent, as presented in Chapters 4–7.

Mainly, the inspection is used to check the quality of requirements. Experienced people remove some of the problems in requirements, but some of the shortcomings remain. Some deficiencies are cleared and corrected or removed gradually during the program development in the following development steps. Some of them are discovered during testing, and some others are revealed after release by the customer.

Mistakes made but not corrected during the first phase of the cycle migrate then to other phases. This fact results in a costly process because all corrections cause costs that increase multiplicative with the later discovery and correction of the bugs. [96]

It is known that manual approaches that rely on human intelligence and the application of inspection checklists do not scale to large specifications [25].

Therefore, we try to detect errors in the requirements specification caused by ambiguity, incompleteness, and inconsistency during the requirements definition process. We provide defect density, i.e., we count the number of detected places in formulations that may have the meanings of defect. A human intervention makes the decision.

In the previous chapters, we described our methods for checking ambiguity, incompleteness, and inconsistency of textual functional requirements specifications. Using these methods, we have defined quality metrics that are used in the process of requirements' quality improvement. Our methods are supported by the implemented tool TEMOS.

8.2 Quality Measurement

Quality in software development on all levels is a topic discussed for many years, and it has been included in ISO standards [65], [43], [15]. Quality of requirements is a specific part of it. It is a well-established concept that is described in many articles.

In [26], the authors list 24 qualities of requirements (we selected only three of them – ambiguity, incompleteness, and inconsistency). Most of them are qualitative – they can only be judged and not be measured. In [26], the authors state a 200:1 ratio between detecting and repairing an error during the requirements specification vs. maintenance phase.

A mapping between requirements specification and a UML model is described in [16], but the author uses controlled English.

In [36], a quality model of requirements is presented that investigates the impact of each of them. In [35], the authors classify 166 rules for requirements and estimate that 53 % of them can be checked automatically with good heuristics. They investigate the problem of what cannot be checked automatically in requirements. We do not exclude human intervention because we think that the semantics may be very complex and that the mistakes of automated checking may be very expensive. In [84], the requirements of agile projects are investigated.

In paper [87], a study of practice is given, and it is stressed that the quality problems of requirements are an important topic. The quality evaluation is done manually during review sessions [102]. A survey of methods is given in [106], [78]. The inspection is described also in [119].

A complexity measure for textual requirements is described in [4]. The measure indicates the amount of actions (and actors) and their relations in requirements. We do not investigate correctness like in [37].

A semantic representation of functional requirements is investigated using methods of information retrieval in [109].

Requirements are usually categorized into functional requirements, non-functional requirements, and quality requirements. We investigate the quality of functional requirements, i.e., the quality of their textual description.

In [10], the authors developed methods of detecting quality violations in a requirements specification called *linguistic triggers*. Besides the problem of incompleteness, an approach to ambiguity detection is also presented.

Patterns belong to the standard technology of text mining. In [31], sentence patterns have been used but for performance requirements, not for functional requirements like in our tool. Using NLP for requirements engineering is analyzed in [13], [25], [131], [40]. Building models from requirements is used for example in [97]. Different from our approach, they do not use it to analyze requirements.

Ambiguity is defined in [26] as the percentage of requirements that have been interpreted in a unique manner by all its human reviewers. The ambiguity of words is investigated in [76].

There are many papers about automated construction of glossaries, e.g., [29], [7], [49], [32]. In [76], a method is proposed to extract a glossary from a set of models automatically. We derived a glossary from the text of requirements. They all are investigating the ambiguity of words [51], [7]. We additionally investigate the ambiguity of sentences in Chapter 4.

In [25], the authors explore potential ambiguity and incompleteness based on the terminology used in different viewpoints. They combine the possibilities of NLP technology with information visualization. Their approach is completely different from our approach.

Two incompleteness metrics of input documents of the requirements specifications are described in [38]. The second one – the backward functional completeness that the paper [38] focuses on – refers to the completeness of a functional requirements specification with respect to the input documents. Contrary to the approach in [38], we do not measure the incompleteness using metrics and quantified results, even though it is a good idea. Using our tool, we generate warning messages only and count their numbers.

In [81], a meta-model approach is used to detect the missing information in a conceptual model. It is also an approach of the class forward functional completeness but at the level of a conceptual model.

In [31], sentence patterns are used to uncover incompleteness with performance requirements. According to the unified model, the performance requirements describe time

behavior, throughput capacity, and cross-cutting. The sentence patterns used in the paper are completely different from our sentence patterns.

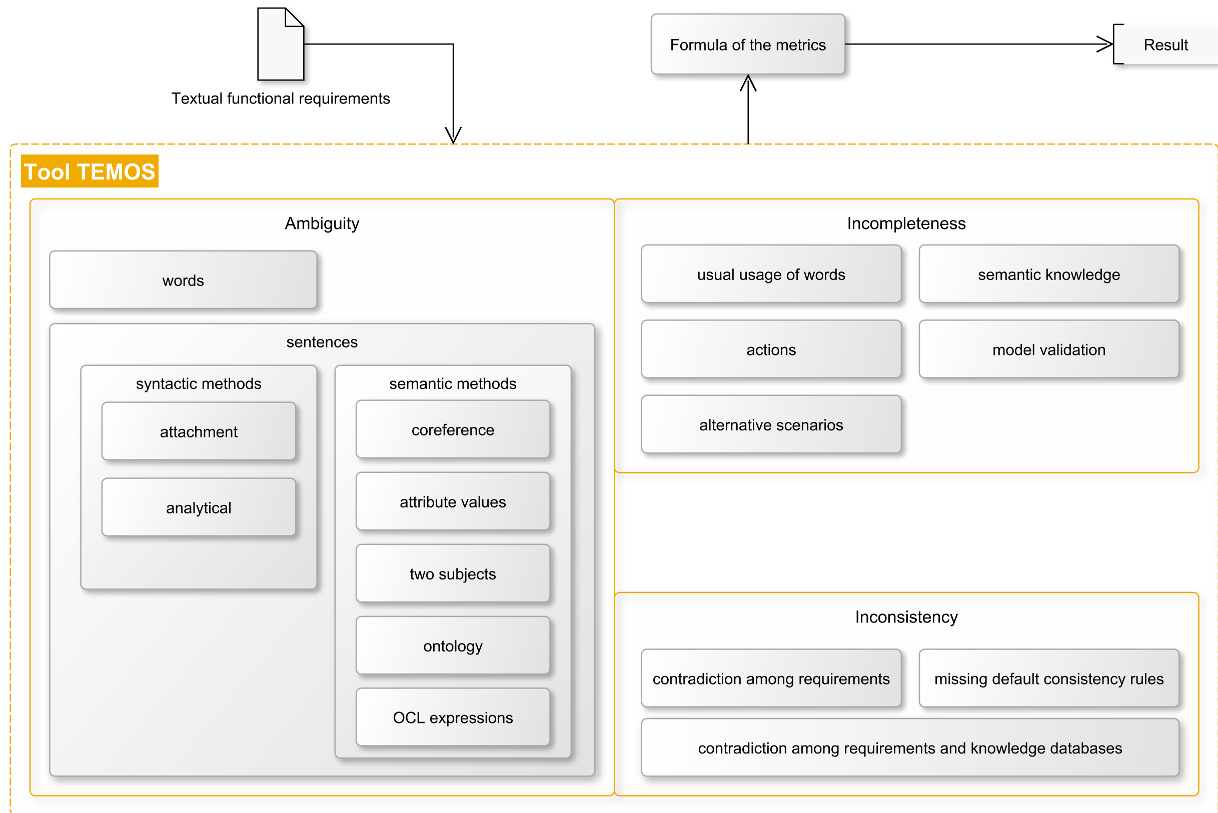


Figure 8.1: Quality evaluation schema.

8.3 Our Approach to Quality of Requirements

We have developed our tool TEMOS to test whether there are some problems of ambiguity, incompleteness, and inconsistency in requirements. Then we found that we can use it to measure the quality of the requirements as a “side effect”.

Our approach is based on a concept that is well-known in traditional publishing houses. During the editing process of manuscripts, all editors’ corrections can be qualified, collected and evaluated for the quality of the manuscript. We used the same method. We present our quality evaluation schema in Fig. 8.1.

Each positive test (i.e., a test revealing a potential problem) is evaluated, and the value becomes part of the quantitative description using metrics. The best quality (zero problems) is achieved when our algorithms do not find any suspicious formulations in the sense of ambiguity, incompleteness, and inconsistency.

While checking, our system TEMOS always generates warning messages when it reveals some suspicious irregularities. We have defined the quality metrics of functional requirements as the value computed from the numbers of the generated warning messages. For clarity, it is structured according to the topics (*ambiguity*, *incompleteness*, *inconsistency*).

In some topics, we can compute the relative quality, which is the number of generated warning messages related to the number of sentences in the requirements.

The proposed metrics use the following components:

- ambiguity
  - AW – the number of ambiguous words found,
  - AS – the number of ambiguous sentences found,
- incompleteness
  - $ISen$ – the number of incomplete sentences found,
  - ISc – the number of incomplete scenarios found,
- inconsistency
  - CGS – the number of contradictions in groups of sentences,
  - DSR – the number of necessary enrichments in the sense of the default consistency rules.

The proposed quality measure formula is

$$\text{Q-Req} = w_1 \cdot AW + w_2 \cdot AS + w_3 \cdot ISen + w_4 \cdot ISc + w_5 \cdot CGS + w_6 \cdot DSR$$

where variables w_i are weights that can be individually set.

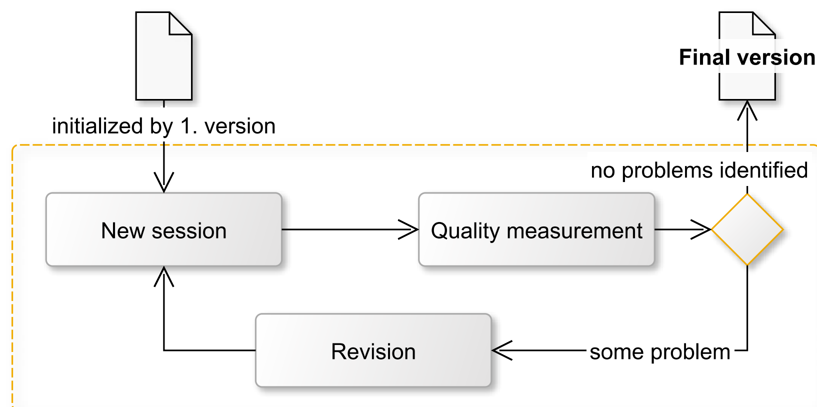


Figure 8.2: Quality measurement iterations as feedback.

As we already mentioned in Section 8.1, our quality measurement can be (and should be) used during the maintenance to master all the inserted corrections, enhancements,

and upgrades requirements into existing textual requirements specification – the process is illustrated in Fig. 8.2.

8.4 Experiments and Results

To evaluate our proposed methods, we reuse the prepared data set[1], which is a collection of textual requirements from four different sources [A.11]. The data set is defined in Section 3.6.1 and we use it for testing our proposed methods in Chapters 4–6.

In Table 8.1, for each input requirements text (case) from the aforementioned data set, we record (a legend of the table):

- the number of all recognized issues (warnings) in the sense of the proposed quality measure formula Q-Req defined in Section 3.6.1 and selected issues measured in the previous chapters, where we use all weights w_i equivalent and equal to 1,
- EN – the number of recognized entities in the requirements text,
- IEN – the number of issues per one recognized entity in the requirements text,
- RN – the number of recognized relations in the requirements text,
- IRN – the number of issues per one recognized relation in the requirements text,
- AN – the number of recognized attributes in the requirements text,
- IPW – the number of issues per word in the requirements text,
- IPS – the number of issues per sentence in the requirements text,
- IPA – the number of issues per average number of words in a sentence in the requirements text.

[1] DOI identifier: `10.5281/zenodo.7897601`

Table 8.1: The evaluation of the recognized issues (warnings) using Q-Req formula.

Case	Issues	EN	IEN	RN	IRN	AN	IPW	IPS	IPA
g02-federalspending	27	81	0.33	51	0.53	0	0.01	0.28	1.27
g03-loudoun	43	34	1.26	23	1.87	1	0.03	0.75	1.55
g04-recycling	25	29	0.86	17	1.47	0	0.02	0.49	1.01
g05-openspending	14	38	0.37	26	0.54	0	0.01	0.26	0.46
g08-frictionless	48	38	1.26	29	1.66	1	0.03	0.73	1.90
g10-scrumalliance	60	62	0.97	45	1.33	1	0.02	0.61	2.29
g11-nsf	25	36	0.69	21	1.19	0	0.01	0.34	1.21
g12-camperplus	36	21	1.71	15	2.40	0	0.03	0.68	1.34
g13-planningpoker	39	41	0.95	34	1.15	1	0.03	0.74	1.42
g14-datahub	26	35	0.74	33	0.79	2	0.01	0.39	0.95
g16-mis	33	63	0.52	62	0.53	1	0.02	0.49	1.46
g17-cask	17	42	0.40	45	0.38	2	0.01	0.27	0.67
g18-neurohub	56	72	0.78	62	0.90	0	0.03	0.55	2.57
g19-alfred	42	55	0.76	42	1.00	2	0.02	0.30	2.37
g21-badcamp	32	39	0.82	26	1.23	0	0.02	0.46	1.19
g22-rdadmp	30	51	0.59	48	0.63	0	0.01	0.36	1.11
g23-archivesspace	14	14	1.00	9	1.56	0	0.02	0.25	0.90
g24-unibath	17	31	0.55	21	0.81	1	0.01	0.33	0.60
g25-duraspace	31	65	0.48	67	0.46	4	0.02	0.31	1.46
g26-racdam	33	33	1.00	23	1.43	0	0.02	0.33	1.54
g27-culrepo	77	94	0.82	64	1.20	1	0.02	0.64	2.65
g28-zooniverse	15	29	0.52	17	0.88	0	0.01	0.25	0.85
P01. Blit	12	30	0.40	34	0.35	1	0.02	0.25	1.10
P02. CS179G – ABC Paint Project	22	95	0.23	83	0.27	6	0.02	0.33	1.21
P03. eProcurement	62	72	0.86	78	0.79	0	0.04	0.69	3.06
P04. Grid 3D	6	18	0.33	13	0.46	0	0.03	0.55	0.37
P05. Home 1.3	25	60	0.42	74	0.34	2	0.02	0.29	1.94
P06. Integrated Library System	124	115	1.08	149	0.83	1	0.06	1.57	4.46

Case	Issues	EN	IEN	RN	IRN	AN	IPW	IPS	IPA
P07. Inventory	116	180	0.64	317	0.37	8	0.02	0.23	13.79
P08. KeePass Password Safe	12	44	0.27	32	0.38	0	0.03	0.33	0.94
P09. Mashbot	14	24	0.58	28	0.50	0	0.02	0.54	0.66
P10. MultiMahjong	62	61	1.02	66	0.94	1	0.04	0.70	3.21
P11. Nenios	70	44	1.59	59	1.19	2	0.07	0.85	6.08
P12. Pontis 5.0 Bridge Management System	71	168	0.42	267	0.27	3	0.02	0.32	3.83
P13. Public Health Information Network	140	181	0.77	244	0.57	0	0.05	1.27	10.68
P14. Publications Management System	96	99	0.97	196	0.49	3	0.04	1.57	3.43
P15. Puget Sound Enhancements	36	104	0.35	123	0.29	1	0.02	0.39	1.90
P16. Tactical Control System	61	223	0.27	204	0.30	1	0.01	0.21	3.10
P17. Tarrant County Integrated Justice IS	79	30	2.63	33	2.39	4	0.04	0.59	6.00
P18. X-38 Fault Tolerant System Services	88	211	0.42	233	0.38	4	0.02	0.25	5.95
H01. CCHIT	65	173	0.38	205	0.32	4	0.03	0.58	3.10
H02. CM1	6	21	0.29	13	0.46	0	0.01	0.20	0.35
H03. InfusionPump	24	186	0.13	185	0.13	2	0.01	0.10	1.61
H04. Waterloo	175	355	0.49	1070	0.16	3	0.01	0.26	9.85
C01. Amazing Lunch Indicator	78	101	0.77	251	0.31	9	0.02	0.84	2.28
C02. EU Rent	26	61	0.43	97	0.27	12	0.05	0.60	2.33
C03. FDP Expanded Clearinghouse Pilot	6	39	0.15	29	0.21	1	0.01	0.15	0.49
C04. Library System	57	74	0.77	109	0.52	1	0.03	0.45	4.01
C05. Nodes Portal Toolkit	16	97	0.16	259	0.06	1	0.01	0.10	1.09
C06. Online National Election Voting	135	93	1.45	137	0.99	3	0.04	0.51	12.42
C07. Restaurant Menu & Ordering System	17	32	0.53	65	0.26	0	0.02	0.37	0.87

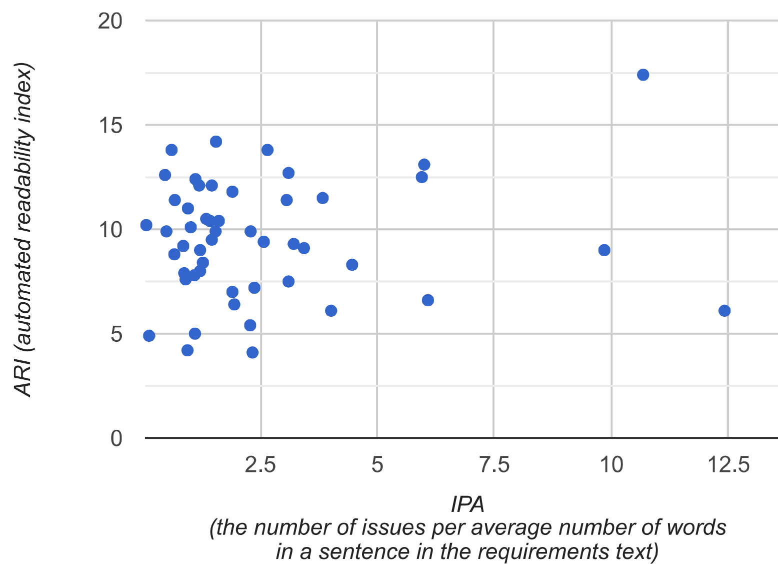


Figure 8.3: Correlation comparison of IPA and ARI.

In Figure 8.3, we compare the correlation of IPA (the number of issues per average number of words in a sentence in the requirements text) and ARI (automated readability index). According to the used data set, it cannot be claimed that the requirements texts with the most generated warnings are the most complex texts according to ARI at the same time.

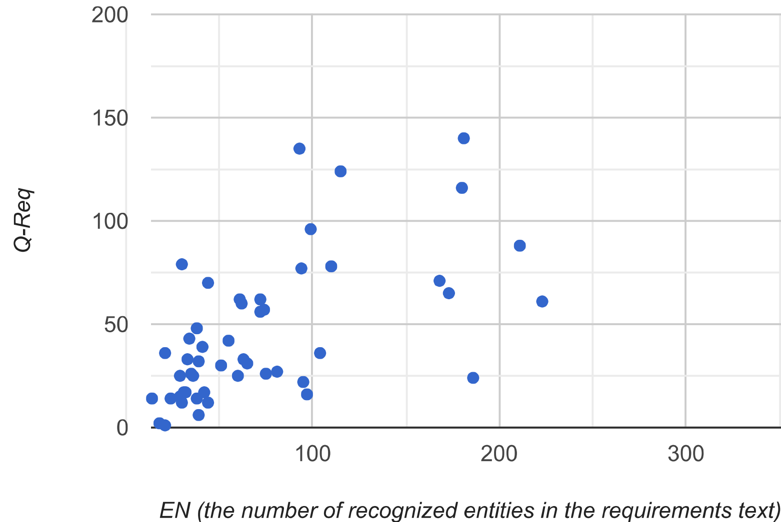


Figure 8.4: Correlation comparison of EN and Q-Req.

In Figure 8.4, we compare the correlation of EN (the number of recognized entities in the requirements text) and the Q-Req formula result (the number of generated warnings). In this case, as might be intuitively expected, the increasing size of the generated model in the sense of recognized entities is reflected in the number of generated warnings due to the need for a more complex specification.

Chapter 9

Created Artifact and Models Generation

This chapter reflects our publications:

- Šenkýř, D.; Suchánek M.; Kroha, P.; Mannaert, H.; Pergl, R. Expanding Normalized Systems from textual domain descriptions using TEMOS. In: *Journal of Intelligent Information Systems*. Springer, 2022. [A.2]
- Šenkýř, D. SHACL Shapes Generation from Textual Documents. In: *Enterprise and Organizational Modeling and Simulation*. Springer, Cham, 2019. [A.7]
- Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies*. SciTePress, Madeira, 2018. [A.10]

This chapter describes the created software artifact – our tool TEMOS. We present features of the implemented tool and ideas on how such a tool could be created. We present used frameworks, too. Finally, we present the benefit of creating the internal model defined in Chapter 3 in the form of exporting the mapped model to various formats.

9.1 TEMOS – Textual Modeling System

As mentioned, TEMOS is an acronym for *Textual Modeling System*. Firstly, we implemented this tool as a prototype to recognize entities and relationships between them for basic statistics of the requirements text, e.g., how many entities the text approximately contains, if all of them have attributes (that can indicate if they are properly described), etc. While investigating the problems described in the previous chapters, we extend TEMOS with the features to detect these problems and highlight them in the text.

Based on the categorization presented in [11] that distinguishes between tools focused on one specific linguistic inaccuracy and tools capable of identifying multiple linguistic inaccuracies, our tool TEMOS belongs to the second category.

Fig. 9.1 shows the screenshot of the tool's main page.

Figure 9.1: TEMOS – main page.

In the following sections, we describe the features of TEMOS. *Disclaimer*: The following sections present feature ideas; they do not present a complete specification. Therefore, the text is incomplete, which may lead to ambiguity.

9.1.1 Core Features

We describe the features primarily from the perspective of the user interface we designed; however, they can be viewed in general as features that a similar system could provide.

First, a user provides the text of requirements and starts the analysis process.

9.1.1.1 Model Recognition

Available features in the mode of the analyzed document are as follows.

- The main output of the analysis process is an annotation of the text of requirements according to the internal model definition (Section 3.4). In our tool, the user interface looks as shown in Figure 9.2.
  - A user can navigate through the text and, according to the annotation, see recognized parts representing entities, attributes, and relationships between entities.
- A user can see all recognized element groups. When a user selects the concrete element group, a list of all elements of the selected element group is provided.
  - A user can merge elements in the concrete element group.
  - A user can display details of the selected element – a list of element attributes and a list of other elements connected to the selected element via a relationship.
- A user can display and manage the glossary.
  - A user can merge terms and corresponding elements using synonym help, as proposed in Chapter 4.
- A user can export the internal model in various formats, as proposed in Section 9.2.

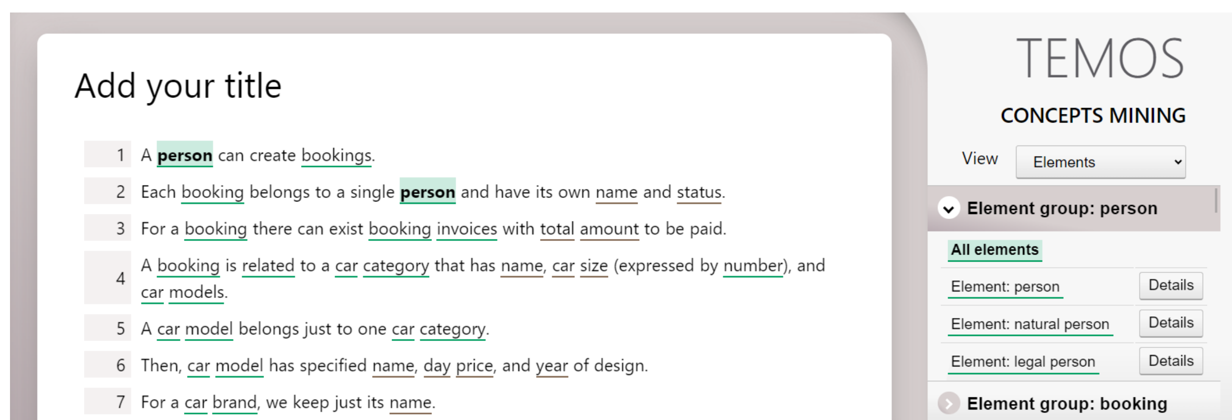


Figure 9.2: Generated elements.

9.1.1.2 Generated Warnings

The analysis process configuration allows a user to select which category of issues should be analyzed (ambiguity, incompleteness, or inconsistency). The recommended way is to analyze all issue type categories.

As a result, a user can see a list of warnings and annotated text where these warnings are highlighted. In our tool, the user interface looks as shown in Figure 9.2.

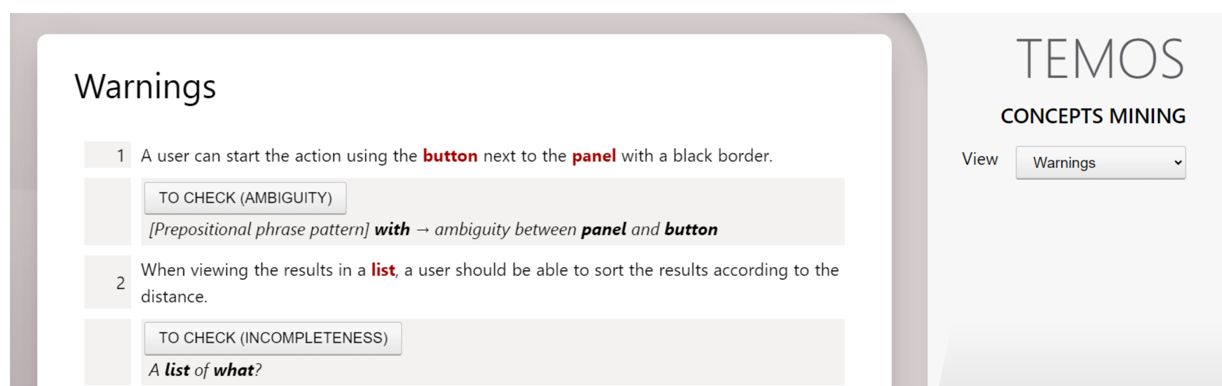


Figure 9.3: Generated warnings concerning ambiguity.

9.1.2 Additional Features

As a by-product, we offer sentence visualization in the form of graph rendering containing separated tokens, their part-of-speech tags, and relationships between tokens. Exactly the same form that we use in our approach based on grammatical inspection, and therefore, we have all relevant data available.

Using the parsing structure, we can also highlight *pronouns*, *passive voice* parts, or *coreference* in the text. The analyst should consider all three to determine whether their presence in the text is appropriate.

9.1.3 Used Technologies

TEMOS tool is represented by the backend core part written in Python, and the frontend part is developed as a single-page application using HTML, CSS, and TypeScript without a specific frontend framework.

TEMOS tool core analyzing part is powered by the *spaCy*[1] NLP framework, in version 3.7.2. We use a pre-trained model called `en_core_web_trf` in version 3.7.3 (available together with the spaCy installation) to process text written in English. The text-processing logic is written in Python. Both the spaCy version and the model version are the latest versions available at the time of writing this thesis. We selected the `en_core_web_trf`

[1] `https://spacy.io`

trained model based on the English models documentation [52], where we compared the accuracy values. The selected model performs the same or better in 11 of 13 categories compared to the second best model (`en_core_web_lg`, which is larger in size.

9.2 Models Generation

Methods detecting ambiguity, incompleteness, and inconsistency are embedded in our tool TEMOS. Many of the presented approaches benefit from the internal model defined in Section 3.4. The internal model semantically represents recognized domain entities, their attributes, and the relationships between entities. The internal model could be further reused for:

- visual feedback and
- further processing in the sense of exporting our model and importing exported data into other tools.

To discuss these two points, we describe the types of generated models (formats) that our tool supports in the following sections.

As a prerequisite, we expect that analysts have processed generated hints and finalized the internal model M (see Section 3.4) in the sense of:

- merge terms according to the glossary processing,
- merge entity elements ($e \in E$) clustered in the element groups (set G) if they represent the same concept, and
- review warnings about hierarchy cycle detection of entity (class) candidates.

When the hints settlement is complete, we can narrow down our internal model to a set of elements E and a set of relationships R ready to be exported.

9.2.1 Generated Models – UML (Class Diagram)

The standard class diagram is handy to visualize the complexity of the requirements. A user can see the number of classes and the relations between them. A user can also see suspicious classes not related to other classes or classes without attributes.

Our tool generates GraphViz code to visualize the diagram easily, and we support XMI and Ecore (a format used in Eclipse Modeling Framework [53]) to standardize the output.

The format of XMI (*XML metadata interchange*), as the name suggests, is an XML structure. The XMI format is a standard of the Object Management Group [92]. The serialized XMI file can be imported for further processing, e.g., in Enterprise Architect [117].

In this section, we show the mapping of our internal model to the XMI format (XML structure). ECore format also uses XML structure (with different tags); therefore, we do not show it here because the mapping is similar. GraphViz uses DOT language for

defining nodes and edges in a graph. Our internal model could be serialized as a graph, where classes represent graph nodes and relationships represent graph edges. DOT format supports HTML elements, so classes could be represented as HTML `table` elements. We do not show details here.

9.2.1.1 Mapping to XMI

For the illustrative purpose of XMI serialization generation, we use the minimalistic requirements of a hotel room booking system shown in Example 9.1. We used the same sentences in the description of suitable patterns in Section 3.5. The sentence structure visualization (in the sense of part-of-speech tags and dependency types) of Example 9.1 is shown in Fig. 3.3, Fig. 3.4, Fig. 3.7, and Fig. 3.8.

Example 9.1: A minimalistic requirements of a hotel room booking system.

A booking is placed by a guest. A booking is related to a room. For a booking, we keep the start date, end date, and total price. A room can be a family apartment or a standard room.

After processing by our tool, the internal model should look like this:

- a set of elements E containing:
  - $E1 \subset E$ having entity (class) candidate type: `Booking`, `Guest`, `Room`, `Family apartment`, and `Standard room`,
  - $E2 \subset E$ having attribute candidate type: `start date`, `end date`, and `total price`, where all attributes candidates belong to the `Booking` entity element (class),
- a set of relationships R containing:
  - r_1 and r_2 representing the association between the `Booking` class and the `Guest` class, and the `Booking` class and the `Room` class, respectively,
  - r_3 and r_4 representing class hierarchy where the `Room` class is the parent class and the `Family apartment` class and the `Standard room` class are subclasses, and
  - relationships representing the attribute ownership.

The abridged result of XMI file generation is in Listing 9.1. We refer to the lines of this listing when describing the following steps used for conversion into XMI format.

1. First, we generate the header (lines 1–6), where we state the XMI specification version and we list TEMOS as the exporter of this file.

2. Next, we generate the model tag (line 7).
3. Then, we generate a root package for the model as a *packaged element* with an ID equal to 0.
4. We keep a number series starting with 1 representing ID for other packaged elements.
5. For each element e_i from the set of elements E:
   - if the *is entity candidate* property equals true, we convert e_i to a *packaged element* with an ID (from the previously mentioned number series in the fourth point) and a name represented by the *full lemma* property of e_i (lines 8–22),
   - otherwise, when e_i is an attribute candidate of entity candidate with an already created *packaged element*, we include it as a child tag called *owned attribute* having the *full lemma* property of e_i as the name (lines 12–13); the visibility is private by default.
6. For each relationship r_j from the set of relationships R:
   a) if the r_j type is not specific, we generate it as a standard association as another *packaged element* that has two ends (tags called *owned end*) referring to the ID of created packaged elements representing classes (lines 24–42); we include recognized multiplicities for both ends of r_j (lines 30–32 and 38–40) – multiplicity "many" is represented with type `uml:LiteralUnlimitedNatural` and value equals -1,
   b) if the r_j type is specific, e.g., represents a *generalization*, we generate a corresponding tag in the corresponding *packaged element*, e.g., a `generalization` tag for classes representing subclasses with an ID referring to the parent class represented by a *packaged element* (line 21).

Listing 9.1: Example of generated XMI file.

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <xmi:XMI xmi:version="2.1"
03:          xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
04:          xmlns:uml="http://schema.omg.org/spec/UML/2.1">
05:   <xmi:Documentation exporter="Textual Modeling System (TEMOS)"
06:                      exporterVersion="1.0" />
07:   <uml:Model xmi:type="uml:Model" name="Hotel_Model">
08:     <packagedElement xmi:type="uml:Package"
09:                      name="Hotel" xmi:id="0">
10:       <packagedElement xmi:type="uml:Class"
11:                        name="Booking" xmi:id="1">
```

```
12:          <ownedAttribute xmi:type="uml:Property"
13:                          name="start date" visibility="private" />
14:          ...
15:        </packagedElement>
16:        <packagedElement xmi:type="uml:Class"
17:                         name="Room" xmi:id="2">
18:        </packagedElement>
19:        <packagedElement xmi:type="uml:Class"
20:                         name="Family apartment" xmi:id="3">
21:          <generalization xmi:type="uml:Generalization" general="2" />
22:        </packagedElement>
23:        ...
24:        <packagedElement xmi:type="uml:Association"
25:                         name="relates">
26:          <memberEnd xmi:idref="4" />
27:          <ownedEnd xmi:type="uml:Property"
28:                    isOrdered="true" xmi:id="4">
29:            <type xmi:idref="1" />
30:            <lowerValue xmi:type="uml:LiteralInteger" value="0" />
31:            <upperValue xmi:type="uml:LiteralUnlimitedNatural"
32:                        value="-1" />
33:          </ownedEnd>
34:          <memberEnd xmi:idref="5" />
35:          <ownedEnd xmi:type="uml:Property"
36:                    isOrdered="true" xmi:id="5">
37:            <type xmi:idref="2" />
38:            <lowerValue xmi:type="uml:LiteralInteger" value="0" />
39:            <upperValue xmi:type="uml:LiteralUnlimitedNatural"
40:                        value="-1" />
41:          </ownedEnd>
42:        </packagedElement>
43:        ...
44:      </packagedElement>
45:    </uml:Model>
46: </xmi:XMI>
```

9.2.2 Generated Models – SHACL

Shapes Constraint Language (SHACL) is the new recommendation by the W3C consortium to uniform both describing and constraining the content of an *RDF graph*. Based on the inspiration of UML class diagram generation from textual requirements specifications in the previous section, we investigate the possibility of mapping parts of a textual document

to *shapes* described by SHACL.

First, we introduce the context, and then we show an illustrative example of SHACL *shapes* generation from our internal model.

9.2.2.1 Motivation – Structured Knowledge on Web

Most, but not all, of the structured knowledge on the Web is deeply connected to the *Semantic Web* and its standards. From history, we can mention the original intention of HTML *meta tags*, which were unfortunately predominantly used for spam – therefore, they are widely ignored by search engines [115].

Nowadays, we can use description-logic-based languages (e.g., OWL, SHACL) provided in the form of a recommendation by the W3C consortium. Based on them, in the second half of the 2000s, projects like *DBpedia* [8], *Freebase*[2], or *Schema.org*[3] started. They represent *knowledge graphs* (ontologies) formed by RDF *triplets*. We can also mention a semantic network called *ConceptNet* [113] that combines its own data with other resources (including the mentioned *DBpedia*) to provide a meaning of words or phrases entered as a query.

Let us briefly introduce the mentioned standards and corresponding technologies.

RDF

The *Resource Description Framework* (RDF) [22] is a W3C specification used as a general approach for the conceptual modeling of information using various syntax notations and data serialization formats. The structure is formed by a set of *triplets* – each consisting of a *subject*, a *predicate*, and an *object*. We benefit from such a structure in our internal model, too, as proposed in Section 3.5.1. The set of *triplets* creates an *RDF graph*.

RDF Schema

The *Resource Description Framework Schema* (RDF Schema or just RDFS) [54] is a semantic extension of RDF. It provides a data-modeling vocabulary for RDF data – a mechanism for describing groups of related resources and the relationships between these resources.

OWL

The W3C *Web Ontology Language* (OWL) [83] is a computational logic-based language. It is perceived as the first level above RDF required for the Semantic Web, which can formally describe the meaning of the terminology used in Web documents. The knowledge expressed in OWL can be exploited by computer programs, e.g., to extend knowledge of the specific problem or to verify the consistency of specifically requested knowledge.

The basic building elements are *classes*, typically arranged in a *subclass hierarchy*. Below, you can find an example (Listing 9.2, in *Turtle* notation) presented in [73]. Note that OWL relies on *RDF Schema vocabulary* for the basic mechanism.

[2] terminated project – data still available via `https://developers.google.com/freebase`

[3] `https://schema.org`

Listing 9.2: Example of Turtle notation (1).

```
@prefix ex: <http://example.com/ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Person
        a owl:Class ;
        rdfs:label "Person" ;
        rdfs:comment "A human being" .

ex:Customer
        a owl:Class ;
        rdfs:subClassOf ex:Person .
```

We introduced the `Customer` subclass of the parent class `Person`. In OWL notation, let us say that no `Person` can have more than one father – Listing 9.3.

Listing 9.3: Example of Turtle notation (2).

```
@prefix ex: <http://example.com/ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Person
        a owl:Class ;
        rdfs:subClassOf [
                a owl:Restriction ;
                owl:onProperty ex:hasFather ;
                owl:maxCardinality 1 ;
        ] ;
        rdfs:subClassOf [
                a owl:Restriction ;
                owl:onProperty ex:hasFather ;
                owl:allValuesFrom ex:Person ;
        ] .
```

OWL operates on classes, which are understood as sets of instances that satisfy the same restrictions. OWL includes the metaclass `owl:Restriction`, which is typically used

as an anonymous superclass of the named class that the restriction is about [73].

SHACL

The *Shapes Constraint Language* (SHACL) is the new recommendation by W3C introduced in July 2017. The purpose of SHACL is to uniform both describing and constraining the content of the *RDF graph*. The sets of constraints used by SHACL for validation are expressed as an RDF graph, and they are called *shapes* or *shape graphs*. The RDF data being validated is called the *data graph*. *Shapes* offer a description of the *data graph* in the form of *constraints* that a valid *data graph* satisfies [93].

Let us continue with Listing 9.3 above. SHACL offers more flexibility in the way of restriction definition. In Listing 9.4, the equivalent of the previous example in the SHACL language is presented.

Listing 9.4: Example of SHACL notation.

```
@prefix ex: <http://example.com/ns#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

ex:Person
        a owl:Class, sh:NodeShape ;
        sh:property [
                sh:path ex:hasFather ;
                sh:maxCount 1 ;
                sh:class ex:Person ;
        ] .
```

Another example of defining SHACL *shapes* is presented in [93]. A nice comparison of built-in constraint types is presented in [73].

9.2.2.2 Mapping to SHACL Shapes

For the illustrative purpose of SHACL *Shapes* generation, we use the minimalistic requirements of a university information system (Example 9.2) that is taken (and shortened) from our paper [A.7].

Example 9.2: A minimalistic requirements of a university information system.

A user is either a student or a teacher. Every user has exactly one username. Each student has at least one subject enrolled.

After processing by our tool, the internal model should look like this:

- a set of elements E containing:
  - $E1 \subset E$ having entity (class) candidate type: `User`, `Student`, `Teacher`, and `Subject`,
  - $E2 \subset E$ having attribute candidate type represented by one element – `username` – belonging to the `User` entity element (class),
- a set of relationships R containing:
  - r_1 and r_2 representing class hierarchy where the `User` class is the parent class and the `Student` class and the `Teacher` class are subclasses,
  - r_3 representing attribute (`username`) ownership with multiplicity equals 1,
  - r_4 representing a relationship between the `Student` class and the `Subject` class with a matched multiplicity.

The result of SHACL *shapes* generation is in Listing 9.5. We refer to the lines of this listing when describing the following steps used for the result format generation. The SHACL validation format options are more complex, as presented in [73]. We focus on a subset of it as follows.

1. First, we generate the header (lines 1–4), where we define the prefixes.
2. For each element e_i from the set of elements E:
   - if the *is entity candidate* property equals true, we convert e_i to a *class node* (lines 9–27),
   - otherwise, when e_i is an attribute candidate of entity candidate with an already created class element, we include it as a child node called `property` (lines 8–12) with multiplicity restrictions if they are recognized.
3. For each relationship r_j from the set of relationships R:
   a) if the r_j type is not specific, we generate it as a standard association as a `property` with recognized multiplicity (lines 17–21),
   b) if the r_j type is specific, e.g., represents a *generalization*, we generate a corresponding statement in the corresponding class shape, e.g., `subClassOf` for shapes representing subclasses (lines 16 and 23).

Listing 9.5: Example of generated SHACL file.

```
01: @prefix ex: <http://example.com/ns#> .
```

```
02: @prefix sh: <http://www.w3.org/ns/shacl#> .
03: @prefix owl: <http://www.w3.org/2002/07/owl#> .
04: @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
05:
06: ex:User
07:     a owl:Class, sh:NodeShape ;
08:         sh:property [
09:             sh:path ex:username ;
10:             sh:minCount 1 ;
11:             sh:maxCount 1 ;
12:         ] .
13:
14: ex:Student
15:     a owl:Class, sh:NodeShape ;
16:         rdfs:subClassOf ex:User ;
17:         sh:property [
18:             sh:path ex:hasSubject ;
19:             sh:minCount 1 ;
20:             sh:class ex:Subject ;
21:         ] .
22: ex:Teacher
23:     a owl:Class, sh:NodeShape .
24:     rdfs:subClassOf ex:User ;
25:
26: ex:Subject
27:     a owl:Class, sh:NodeShape .
```

9.2.3 Generated Models – Normalized Systems

In [A.2], we transform our internal model M (see Section 3.4) into a model of Normalized Systems elements. This may realize a text-to-software pipeline. The input is represented by textual requirements processed by our tool TEMOS. Its output is a running prototype of an information system created using Normalized Systems techniques.

To achieve this, we use the phases as follows.

9.2.3.1 Mapping to an NS Metamodel

When the input text is processed, our internal model M is ready for conversion to an NS model. The following steps are used for conversion.

1. For each element e_i from the set of elements E in our model M:
    - if the *is entity candidate* property equals true, we convert e_i to a *data element* in the created NS model,
    - otherwise, when e_i is an attribute candidate, we assign it as a *value field* to the corresponding *data element* representing the entity.
2. The type of data element is *primary* by default. However, if the root lemma of e_i is *type*, then the data element type is *taxonomy*.

3. We convert each binary relation (relationship) r_j from the set of relationships R in our model M to a *link field* of the data element created from e_k (in the first step). We set the *link field* properties as follows:

    - *link field type* is set based on the multiplicity type of r_j,
    - *required* is also set based on the multiplicity type of r_j,
    - *target* is set to the data element e_l already created (in the first step).

4. We convert each recognized hierarchy relation. According to NS theory, inheritance causes combinatorial effects; i.e., it is considered an obstacle to evolvability. Therefore, inheritance must be modeled using link fields. These link fields are created based on the data element representing subentity e_k of parent entity e_l. The target of a link field is the data element representing e_l. In this case, the required property always equals *true*, and the multiplicity is *singular*.

5. The last step is the enhancement phase. We describe it in the next section.

9.2.3.2 Enhancement Phase

After composing a model of data elements, we add several steps to apply conventions from NS modeling and enhance the resulting model. The first step is to handle relations between the primary data elements and their corresponding taxonomy data elements. It is a pervasive pattern to have a taxonomy data element with name suffix *Type* and just a single attribute *name*. In other modeling languages, such as UML, this would become an enumeration. However, in terms of NS theory, enumerations block evolvability (e.g., when we need to add fields to enumerated items). We note that the type is specified as only a “type” without any additional information in the textual requirements. When such a value field is created, it is changed in the enhancement phase to a link field linking the newly created taxonomy data element with the name suffix *Type* and a *name* value field. For example, if there is a `Vehicle` data element with a `type` value field, it is changed to a link field pointing to a new `VehicleType` that has a `name`.

The second step is again related to the taxonomy data elements. We note that some texts describe both primary and taxonomy data elements but not the relation between them. This is addressed by simply looking for such data elements with missing relations to the corresponding taxonomy data element (if they exist). For each of them, a new link field in the primary data element is created. The final step of enhancement adjusts the link field names using the conventions of NS modeling. According to the mapping, the names of the fields are taken from triples that do not allow matching bidirectional links in NS models. If there is just a single link field for the target data element, it is named with the target element’s name. In the case of a many-sided relationship, i.e., a relationship that contains a collection of target element instances, a suffix is added to express plurality. For example, the value field `drives` from *“Person drives vehicles”* is renamed `vehicles`, but we keep *drives* as a description of the field.

9.2.3.3 Exporting NS Elements

The classes for representing data elements, their value and link fields, and other properties are expanded directly from the NS metamodel, which is meta-circular. The XML format is required to allow the translation of NS models in TEMOS to NS models for expanders. Due to the versatility of NS tooling, we are able to implement expanders for the data classes in Python and XML serialization. There are two benefits to using the approach with expanders in this case. First, when the NS metamodel is updated, it is possible to re-expand the classes and related serialization for TEMOS. Second, although we currently focus on the structural part of the NS metamodel (i.e., data elements and the constructs around them), expansion works with a full NS metamodel. It will simplify future development when we can focus, for example, on task and flow elements. TEMOS has been extended with a command-line interface that takes text as input and writes a set of XML files with data elements according to the NS tooling expectations.

The set of XML files generated by TEMOS can be added to an existing NS component or to a newly prepared component. It is not possible to generate a component completely from functional requirements. A component contains various implementation and environment details, e.g., the source base used or the version or qualified name of the component. These details need to be provided in other NS tools, or an example component XML file can be adjusted accordingly. The component can then be imported for further refinement, e.g., in NS Modeler, or directly used for expansion.

We reused the minimalistic requirements of a hotel room booking system from Example 9.1 to generate an NS model of data elements. A more detailed example is presented in our paper [A.2].

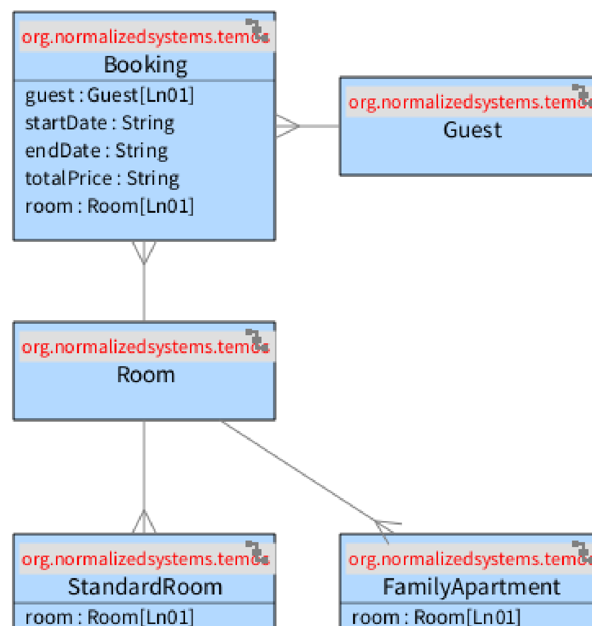


Figure 9.4: Generated data elements for a hotel room booking system example.

Fig. 9.4 shows the resulting model with five data elements. There are no fields for the `Guest` data element and the `Room` data element mentioned in the text. Both `StandardRoom` and `FamilyApartment` data elements have a relation to the parent element `Room`, as they form a hierarchy. Finally, there are several fields (value and link) for the `Booking` data element.

9.2.4 Generating Information System Prototypes

Expansion is directly possible from a Normalized Systems component filled by data elements from our transformation. The expanded information system can be built and deployed by simply executing a set of commands or by clicking the button in the Normalized Systems tool called Prime Radiant [88]. The resulting system is a basic CRUD application but is fully operational and can serve as a prototype for further elaboration on requirements. Fig. 9.5 shows a form based on the `Booking` data element. We manually added the value field `name` for `Room` and `Guest`, as data elements without any attribute are meaningless – we discuss this issue in Section 5.3.5.

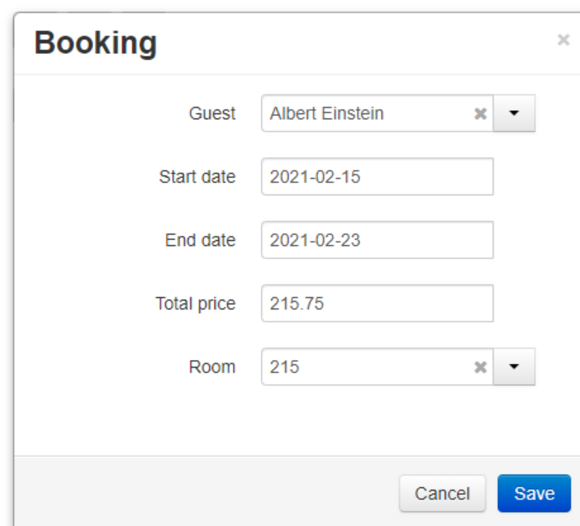


Figure 9.5: Booking form for expanded NS application.

With the prototype, it is possible to change the NS model directly in NS Modeler or Prime Radiant and quickly re-expand, build, and redeploy the application. If the requirements are more specific, even custom code fragments called craftings can be added to the expanded code base. NS comes with a method of harvesting craftings, so they are not lost upon re-expansion. Whenever there is a change in the requirements text, it is possible to regenerate the XML files of data elements, but if other changes are made, they are overwritten. On the other hand, if there are harvested craftings or elements other than data elements in the component, these additional elements stay intact.

Chapter **10**

Conclusions

10.1 Research Goals Revisited

In Section 1.3, we have defined the following three research goals.

1. **G1.** Identify the type of problems introduced by:
    a) **G1.A** ambiguity,
    b) **G1.B** incompleteness, and
    c) **G1.C** inconsistency.
2. **G2.** Propose algorithms how to identify them in the text.
3. **G3.** Propose method how an analyst could improve the text.

First, we needed to tackle the problem of the text's meaning itself. We described how we use a natural language processing framework to identify sentences, tokens, and different types of dependencies between tokens in Chapter 3. In the same chapter, we introduced our internal model representing the semantics of the text.

Regarding the first research goal **G1**, we devoted a separate chapter to each problem. We described the problems of ambiguity in Chapter 4, the problems of incompleteness in Chapter 5, and we divided the problems of inconsistency into two areas – inconsistency presented in the text itself (Chapter 6) and default consistency rules not presented in the text (Chapter 7). In the same chapters, we also propose methods on how to identify the described problems in the textual requirements via a combination of syntactic methods, semantic methods, and online dictionaries to fulfill the second research goal **G2**.

In Chapter 8, we described the iterative process of how users could interpret generated warnings regarding the identified issues of ambiguity, incompleteness, and inconsistency. This chapter fulfills the third research goal **G3**.

10.2 Contributions of the Dissertation Thesis

All goals of this thesis were achieved. Let us briefly highlight the main contributions of the presented dissertation thesis and our published papers as follows.

1. We provide a list of problems in textual requirements specification in different areas:
    - *ambiguity*:
        - on the level of words,
        - on the sentence level,
    - *inconsistency*:
        - on the sentence level,
        - on the whole document level,
        - on the level of statements not presented in a document (we call them *default consistency rules*),
    - *incompleteness*:
        - on the sentence level,
        - on the level of scenarios.
2. We provide algorithms recognizing problems from the previous point using *natural language processing techniques* and a model created during textual requirements specification processing.
3. We proposed how to extend and validate a created model using *online semantic networks* (e.g., ConceptNet) and *online dictionaries* (e.g., Wordnik).
4. We tested the proposed methods using documents from different data sets, and we proposed the *quality measurement formula*.
    - We published the collected documents as a data set available in Zotero [A.11].
5. We developed a prototype system TEMOS[1] based on the proposed methods to demonstrate how the methods could be used by analysts or other users in the industry.

[1]During development, we reported the problems we found in used resources as feedback.
https://github.com/explosion/spaCy/issues/10699
https://github.com/explosion/spaCy/issues/11301
https://github.com/commonsense/conceptnet5/issues/325

10.3 Future Work

In our work, we have proposed methods and a tool for analysts and other team roles processing the textual requirements of a product. It would be interesting to monitor the usage of such a tool and improve current recognition methods. Similar to the spelling and grammar-check tools, it would be useful to popularize the usage of similar tools to our proposed one to check the model interpreted in the text.

The implementation of our proposed methods could be further improved to recognize selected dynamic models of UML.

Nowadays, tools are available to extract the (UML) model from the code. Although the model of an implemented product has more types of classes (e.g., classes representing internal logic or framework classes), there should be classes representing concepts described in the requirements (especially for domain-driven development). There is an open space for comparing the subset of the model from implementation with the model extracted from the textual requirements and marking the fulfillment (in the testing phase) or the progress (in the development phase).

Considering the dynamic models and the current research on test generation from the text of requirements as proposed in [42], it would be interesting to investigate this area using patterns and our internal domain model recognition method, too.

Bibliography

[1] Esra A. Abdelnabi, Abdelsalam M. Maatuk, and Mohammed Hagal. Generating UML Class Diagram from Natural Language Requirements: A Survey of Approaches and Techniques. In *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, pages 288–293, 2021. doi:10.1109/MI-STA52233.2021.9464433.

[2] Eneko Agirre and Philip Edmonds. *Foundations of Computational Linguistics*. 978-1-4020-4808-4, 1 edition, 2007.

[3] Vincenzo Ambriola and Vincenzo Gervasi. Processing Natural Language Requirements. In *Proceedings of the 12th International Conference on Automated Software Engineering (Formerly: KBSE)*, ASE '97, pages 36–45, Washington, DC, USA, 1997. IEEE Computer Society Press. URL: https://dl.acm.org/citation.cfm?id=786767.786786, doi:10.1109/ASE.1997.632822.

[4] Vard Antinyan, Miroslaw Staron, Anna Sandberg, and Jörgen Hansson. A Complexity Measure for Textual Requirements. In *2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 148–158, Los Alamitos, CA, USA, 2016. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/IWSM-Mensura.2016.030, doi:10.1109/IWSM-Mensura.2016.030.

[5] Seth Appleman, Patrick Coffey, David Kelley, and Cliff Yip. System Requirements Specification – E-Voting. https://docplayer.net/24435423-System-requirements-specification-e-voting-authored-by-seth-appleman-patrick-coffey-david-kelley-cliff-yip.html, 2006. Accessed: 2023-09-24.

[6] Jim Arlow and Ila Neustadt. *UML 2.0 and The Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley, 2nd edition, 2005.

[7] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Automated extraction and clustering of requirements glossary terms. *IEEE Transactions on Software Engineering*, 43(10):918–945, 2017. doi:10.1109/TSE.2016.2635134.

[8] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A Nucleus for a Web of Open Data. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] Frederik S. Bäumer and Michaela Geierhos. Running Out of Words: How Similar User Stories Can Help to Elaborate Individual Natural Language Requirement Descriptions. In Giedre Dregvaite and Robertas Damasevicius, editors, *Information and Software Technologies*, volume 639, pages 549–558. Springer International Publishing, Cham, 2016. URL: https://link.springer.com/10.1007/978-3-319-46254-7_44, doi:10.1007/978-3-319-46254-7_44.

[10] Frederik S. Bäumer and Michaela Geierhos. Flexible Ambiguity Resolution and Incompleteness Detection in Requirements Descriptions via an Indicator-Based Configuration of Text Analysis Pipelines. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, pages 5746–5755, 2018.

[11] Frederik S. Bäumer, Joschka Kersting, and Michaela Geierhos. Natural Language Processing in OTF Computing: Challenges and the Need for Interactive Approaches. *Computers*, 8(1):14, 2019. doi:10.3390/computers8010022.

[12] Wahiba Ben Abdessalem Karaa, Zeineb Ben Azzouz, Aarti Singh, Nilanjan Dey, Amira S. Ashour, and Henda Ben Ghazala. Automatic Builder of Class Diagram (ABCD): An Application of UML Generation from Functional Requirements. *Software: Practice and Experience*, 46(11):1443–1458, 2016. doi:10.1002/spe.2384.

[13] Daniel M. Berry, Erik Kamsties, and Michael M. Krieger. From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity: A Handbook (version 1.0). https://cs.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf, 2003. Accessed: 2023-11-28.

[14] James Billson. What are some good examples of a software requirements specification? https://www.quora.com/What-are-some-good-examples-of-a-software-requirements-specification/answer/James-Billson, 2021. Accessed: 2023-04-18.

[15] Jørgen Bøegh. A New Standard for Quality Requirements. *IEEE Software*, 25(2):57–63, 2008. doi:10.1109/MS.2008.30.

[16] Yegor Bugayenko. Combining Object-Oriented Paradigm and Controlled Natural Language for Requirements Specification. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Beyond Code: No Code*, BCNC 2021, pages 11—-17, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3486949.3486963.

[17] Davide Buscaldi and Paulo Rosso. A Conceptual Density-Based Approach for the Disambiguation of Toponyms. *International Journal of Geographical Information Science*, 22(3):301–313, 2008. doi:10.1080/13658810701626251.

[18] Johnathan Mauricio Calle Gallego and Carlos Mario Zapata Jaramillo. QUARE: towards a question-answering model for requirements elicitation. *Automated Software Engineering*, 30(2):25, 2023. doi:10.1007/s10515-023-00386-w.

[19] Aurek Chattopadhyay, Ganesh Malla, Nan Niu, Tanmay Bhowmik, and Juha Savolainen. Completeness of Natural Language Requirements: A Comparative Study of User Stories and Feature Descriptions. In *2023 IEEE 24th International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 52–57, 2023. doi:10.1109/IRI58017.2023.00017.

[20] Chord (music): Definition. https://en.wikipedia.org/wiki/Chord_(music), 2004. Accessed: 2020-12-21.

[21] Chordfind: 4-String Version. http://chordfind.com/4-string, 2003. Accessed: 2020-09-30.

[22] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. W3C recommendation, W3C, 2014. Available from: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225.

[23] Alberto Rodrigues da Silva and João Costa Fernandes. Variability Specification and Resolution of Textual Requirements. In *Proceedings of the 20th International Conference on Enterprise Information Systems*, pages 157–168. SciTePress – Science and Technology Publications, 2018. doi:10.5220/0006810801570168.

[24] Fabiano Dalpiaz, Ivor van der Schalk, Sjaak Brinkkemper, Fatma Başak Aydemir, and Garm Lucassen. Detecting terminological ambiguity in user stories: Tool and experimentation. *Information and Software Technology*, 110:3–16, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0950584918300715, doi:https://doi.org/10.1016/j.infsof.2018.12.007.

[25] Fabiano Dalpiaz, Ivor van der Schalk, and Garm Lucassen. Pinpointing Ambiguity and Incompleteness in Requirements Engineering via Information Visualization and NLP. In Erik Kamsties, Jennifer Horkoff, and Fabiano Dalpiaz, editors, *Requirements Engineering: Foundation for Software Quality*, pages 119–135, Cham, 2018. Springer International Publishing.

[26] Alan Davis, Scott Overmyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, and Mary Theofanos. Identifying and Measuring Quality in a Software Requirements Specification. In *Proceedings First International Software Metrics Symposium*, pages 141–152, Los Alamitos, CA, USA, 1993. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/METRIC.1993.263792`, `doi:10.1109/METRIC.1993.263792`.

[27] Marie-Catherine de Marneffe, Anna N. Rafferty, and Christopher D. Manning. Finding Contradictions in Text. In Johanna Moore, Simone Teufel, James Allan, and Sadaoki Furui, editors, *Proceedings of ACL-08: HLT*, pages 1039–1047, Columbus, Ohio, 2008. Association for Computational Linguistics.

[28] Jeremy Dick, Elizabeth Hull, and Ken Jackson. *Requirements Engineering*. Springer Cham, 4th edition, 2017. ISBN 978-3-319-61072-6. `doi:https://doi.org/10.1007/978-3-319-61073-3`.

[29] Anurag Dwarakanath, Roshni R. Ramnani, and Shubhashis Sengupta. Automatic Extraction of Glossary Terms from Natural Language Requirements. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 314–319, 2013. `doi:10.1109/RE.2013.6636736`.

[30] Steve Easterbrook. Lecture 17: Requirements Specifications. `https://www.cs.toronto.edu/~sme/CSC340F/slides/17-specifications.pdf`, 2004. Lecture notes, Requirements Engineering (CSC340F), University of Toronto. Accessed: 2023-08-24.

[31] Jonas Eckhardt, Andreas Vogelsang, Henning Femmer, and Philipp Mager. Challenging Incompleteness of Performance Requirements by Sentence Patterns. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 46–55, Beijing, China, 2016. IEEE Computer Society Press. URL: `https://ieeexplore.ieee.org/document/7765510`, `doi:10.1109/RE.2016.24`.

[32] Saad Ezzini, Sallam Abualhaija, Chetan Arora, Mehrdad Sabetzadeh, and Lionel C. Briand. Using Domain-Specific Corpora for Improved Handling of Ambiguity in Requirements. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1485–1497, Los Alamitos, CA, USA, 2021. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00133`, `doi:10.1109/ICSE43902.2021.00133`.

[33] Fabrizio Fabbrini, Mario Fusani, Stefania Gnesi, and Giuseppe Lami. An Automatic Quality Evaluation for Natural Language Requirements. In *Proceedings of the Seventh International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 1, 2001.

[34] Gauthier Fanmuy, Anabel Fraga, and Juan Llorens. Requirements Verification in the Industry. In Omar Hammami, Daniel Krob, and Jean-Luc Voirin, editors, *Complex*

Systems Design & Management, pages 145–160, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[35] Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. Rapid Quality Assurance with Requirements Smells. *Journal of Systems and Software*, 123:190–213, January 2017. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0164121216000789`, `doi:10.1016/j.jss.2016.02.047`.

[36] Henning Femmer, Jakob Mund, and Daniel Méndez Fernández. It's the Activities, Stupid! A New Perspective on RE Quality. In *2015 IEEE/ACM 2nd International Workshop on Requirements Engineering and Testing (RET)*, pages 13–19, Los Alamitos, CA, USA, 2015. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/RET.2015.11`, `doi:10.1109/RET.2015.11`.

[37] Shiling Feng, Xiaohong Chen, Qin Li, and Yongxin Zhao. RE2B: Enhancing Correctness of Both Requirements and Design Models. In *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 191–198, Los Alamitos, CA, USA, 2021. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/TASE52547.2021.00034`, `doi:10.1109/TASE52547.2021.00034`.

[38] Alessio Ferrari, Felice dell'Orletta, Giorgio Oronzo Spagnolo, and Stefania Gnesi. Measuring and Improving the Completeness of Natural Language Requirements. In Camille Salinesi and Inge van de Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, pages 23–38, Cham, 2014. Springer International Publishing.

[39] Alessio Ferrari, Giorgio Oronzo Spagnolo, and Stefania Gnesi. PURE: A Dataset of Public Requirements Documents. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 502–505, 2017. `doi:10.1109/RE.2017.29`.

[40] Alessio Ferrari, Liping Zhao, and Waad Alhoshan. NLP for Requirements Engineering: Tasks, Techniques, Tools, and Technologies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 322–323, Los Alamitos, CA, USA, 2021. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/ICSE-Companion52605.2021.00137`, `doi:10.1109/ICSE-Companion52605.2021.00137`.

[41] Donald Firesmith. Are Your Requirements Complete? *Journal of Object Technology*, 4(1):27–43, 2005.

[42] Jannik Fischbach, Julian Frattini, Andreas Vogelsang, Daniel Mendez, Michael Unterkalmsteiner, Andreas Wehrle, Pablo Restrepo Henao, Parisa Yousefi, Tedi Juricic, Jeannette Radduenz, and Carsten Wiecher. Automatic creation of acceptance tests by extracting conditionals from requirements: NLP approach and

case study. *Journal of Systems and Software*, 197:111549, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0164121222002254, doi:https://doi.org/10.1016/j.jss.2022.111549.

[43] Institute for Electric and Electronics Engineers. IEEE Guide to Software Requirements Specifications, 1984.

[44] Xavier Franch, Cristina Palomares, Carme Quer, Panagiota Chatzipetrou, and Tony Gorschek. The state-of-practice in requirements specification: an extended interview study at 12 companies. *Requirements Engineering*, 28(3):377–409, 2023. doi:10.1007/s00766-023-00399-7.

[45] Edwin Friesen, Frederik S. Bäumer, and Michaela Geierhos. CORDULA: Software Requirements Extraction Utilizing Chatbot as Communication Interface. In Klaus Schmid, Paola Spoletini, Eya Ben Charrada, Yoram Chisik, Fabiano Dalpiaz, Alessio Ferrari, Peter Forbrig, Xavier Franch, Marite Kirikova, Nazim Madhavji, and et al.Editors, editors, *Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018)*, pages 234–241. CEUR-WS.org, 2018.

[46] Francesco Gargiulo, Gabiella Gigante, and Massimo Ficco. A Semantic Driven Approach for Requirements Consistency Verification. *Int. J. High Perform. Comput. Netw.*, 8(3):201–211, August 2015. doi:10.1504/IJHPCN.2015.071261.

[47] Sarah Geagea, Sheng Zhang, Niclas Sahlin, Faegheh Hasibi, Farhan Hameed, Elmira Rafiyan, and Magnus Ekberg. Software Requirements Specification: Amazing Lunch Indicator. http://www.cse.chalmers.se/~feldt/courses/reqeng/examples/srs_example_2010_group2.pdf, 2010. Accessed: 2019-01-21.

[48] Michaela Geierhos, Sabine Schulze, and Frederik S. Bäumer. What Did You Mean? Facing the Challenges of User-generated Software Requirements. In *Proceedings of the International Conference on Agents and Artificial Intelligence*, pages 277–283, Lisbon, Portugal, 2015. SCITEPRESS – Science and and Technology Publications. doi:10.5220/0005346002770283.

[49] Tim Gemkow, Miro Conzelmann, Kerstin Hartig, and Andreas Vogelsang. Automatic Glossary Term Extraction from Large-Scale Requirements Specifications. In *IEEE 26th International Requirements Engineering Conference*, pages 412–417, 2018.

[50] Gitnux. The most surprising software project failure statistics and trends in 2023. https://blog.gitnux.com/software-project-failure-statistics, 2023. Accessed: 2023-06-19.

[51] Benedikt Gleich, Oliver Creighton, and Leonid Kof. Ambiguity Detection: Towards a Tool Explaining Ambiguity Sources. In David Hutchison, Takeo Kanade, Josef

Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Roel Wieringa, and Anne Persson, editors, *Requirements Engineering: Foundation for Software Quality*, volume 6182, pages 218–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-14192-8_20.

[52] ExplosionAI GmbH. English · spacy Models Documentation. https://spacy.io/models/en. Accessed: 2023-11-29.

[53] Richard Gronback. Eclipse Modeling Framework (EMF). https://eclipse.dev/modeling/emf. Accessed: 2023-08-26.

[54] Ramanathan Guha and Dan Brickley. RDF Schema 1.1. W3C recommendation, W3C, 2014. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[55] The Atlantic Systems Guild. Volere Requirements Specification Template. http://www.volere.co.uk/template.htm. Accessed: 2016-11-28.

[56] Jane Huffman Hayes, Jared Payne, and Mallory Leppelmeier. Toward Improved Artificial Intelligence in Requirements Engineering: Metadata for Tracing Datasets. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 256–262, 2019. doi:10.1109/REW.2019.00052.

[57] Graeme Hirst. *Semantic Interpretation and the Resolution of Ambiguity.* Cambridge University Press, 1987.

[58] Florentina Hristea. *The Naive Bayes Model for Unsupervised Word Sense Disambiguation: Aspects Concerning Feature Selection.* Springer Publishing Company, Incorporated, 2012.

[59] Carlos Huertas and Reyes Juárez-Ramírez. NLARE, A Natural Language Processing Tool for Automatic Requirements Evaluation. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 371–378, New York, NY, USA, 2012. ACM Press. doi:10.1145/2381716.2381786.

[60] Carlos Huertas and Reyes Juárez-Ramírez. Towards Assessing the Quality of Functional Requirements Using English/Spanish Controlled Languages and Context Free Grammar. In *The Third International Conference on Digital Information and Communication Technology and its Applications (DICTAP2013)*, pages 234–241, 2013.

[61] Anthony Hunter and Bashar Nuseibeh. Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Trans. Softw. Eng. Methodol.*, 7(4):335–367, October 1998. doi:10.1145/292182.292187.

[62] John Hutchins. Retrospect and Prospect in Computer-Based Translation. In *Proceedings of Machine Translation Summit VII 99*, pages 30–34, Tokyo, 1999. Asia-Pacific Association for Machine Translation (AAMT).

[63] IEEE International Standard for System, Software, and Hardware Verification and Validation. *IEEE Standard 1012-2016 (Revision of IEEE Standard 1012-2012)*, 2016. doi:10.1109/IEEESTD.2017.8055462.

[64] IEEE Standard for Application and Management of the Systems Engineering Process. *IEEE Standard 1220-2005 (Revision of IEEE Standard 1220-1998)*, 2005. doi:10.1109/IEEESTD.2005.96469.

[65] ISO/IEC/IEEE International Standard – Systems and software engineering – Life cycle processes – Requirements engineering. *ISO/IEC/IEEE 29148:2011(E)*, 2011. doi:10.1109/IEEESTD.2011.6146379.

[66] ISO/IEC/IEEE International Standard – Systems and software engineering – Life cycle processes – Requirements engineering. *ISO/IEC/IEEE 29148:2018(E)*, 2018. doi:10.1109/IEEESTD.2018.8559686.

[67] IEEE Recommended Practice for Software Requirements Specifications, 1998. doi:10.1109/IEEESTD.1998.88286.

[68] Grammarly Inc. My Grammarly. https://app.grammarly.com/, 2009–2023. [software: web application]. Accessed: 2023-11-15.

[69] Prathamesh Ingle. Best Natural Language Processing (NLP) Tools/Platforms (2023). https://www.marktechpost.com/2023/04/14/top-natural-language-processing-nlp-tools-platforms, 2023. Accessed: 2023-09-06.

[70] Information technology – Object Management Group Unified Modeling Language (OMG UML), Infrastructure. https://www.omg.org/spec/UML/ISO/19505-1/PDF, 2012. ISO/IEC 19505-1:2012(E). Accessed 2023-09-10.

[71] Massila Kamalrudin and Safiah Sidek. A Review on Software Requirements Validation and Consistency Management. *International Journal of Software Engineering and Its Applications*, 9(10):39–58, 2015.

[72] Mohamad Kassab, Colin Neill, and Phillip Laplante. State of Practice in Requirements Engineering: Contemporary Data. *Innovations in Systems and Software Engineering*, 10(4):235–241, 2014. doi:10.1007/s11334-014-0232-4.

[73] Holger Knublauch. SHACL and OWL Compared. https://spinrdf.org/shacl-and-owl.html. Last updated: 2017-08-17. Accessed: 2019-01-08.

[74] Leonid Kof. An Application of Natural Language Processing to Domain Modelling: Two Case Studies. *International Journal on Computer Systems Science Engineering*, 20:37–52, 2004.

[75] Leonid Kof. Natural Language Processing: Mature Enough for Requirements Documents Analysis? In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Andrés Montoyo, Rafael Muńoz, and Elisabeth Métais, editors, *Natural Language Processing and Information Systems*, volume 3513, pages 91–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. URL: https://link.springer.com/10.1007/11428817_9, doi:10.1007/11428817_9.

[76] Salih G. Köse and Fatma B. Aydemir. Automated Glossary Extraction from Collaborative Requirements Models. In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pages 11–15, Los Alamitos, CA, USA, 2021. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/REW53955.2021.00008, doi:10.1109/REW53955.2021.00008.

[77] Petr Kroha, Robert Janetzko, and José Emilio Labra. Ontologies in Checking for Inconsistency of Requirements Specification. In *2009 Third International Conference on Advances in Semantic Processing*, pages 32–37, Sliema, Malta, 2009. IEEE Computer Society Press. URL: https://ieeexplore.ieee.org/document/5291538, doi:10.1109/SEMAPRO.2009.11.

[78] Patrick Sebastian Kummler, Léa Vernisse, and Hansjörg Fromm. How Good are My Requirements? A New Perspective on the Quality Measurement of Textual Requirements. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 156–159, Los Alamitos, CA, USA, 2018. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/QUATIC.2018.00031, doi:10.1109/QUATIC.2018.00031.

[79] Oi Yee Kwong. *New Perspectives on Computational and Cognitive Strategies for Word Sense Disambiguation*. Springer Publishing Company, Incorporated, 2013.

[80] Mathias Landhäußer, Sven J. Körner, and Walter F. Tichy. From Requirements to UML Models and Back: How Automatic Processing of Text Can Support Requirements Engineering. *Software Quality Journal*, 22(1):121–149, 2014. doi:10.1007/s11219-013-9210-6.

[81] Ao Li. Analysis of Requirements Incompleteness Using Metamodel Specification. Master's thesis, University of Tampere, June 2015.

[82] Mich Luisa, Franch Mariangela, and Novi Inverardi Pierluigi. Market Research for Requirements Analysis Using Linguistic Tools. *Requirements Engineering*, 9(1):40–56, 2004. doi:10.1007/s00766-003-0179-8.

[83] Deborah McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C recommendation, W3C, 2004. Available from: https://www.w3.org/TR/2004/REC-owl-features-20040210.

[84] Juliana Medeiros, Miguel Goulão, Alexandre Vasconcelos, and Carla Silva. Towards a Model about Quality of Software Requirements Specification in Agile Projects. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 236–241, Los Alamitos, CA, USA, 2016. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/QUATIC.2016.058, doi:10.1109/QUATIC.2016.058.

[85] Juliana Medeiros, Alexandre Vasconcelos, Carla Silva, and Miguel Goulão. Requirements specification for developers in agile projects: Evaluation by two industrial case studies. *Information and Software Technology*, 117, 2020. doi:https://doi.org/10.1016/j.infsof.2019.106194.

[86] Roberto Navigli and Simone Paolo Ponzetto. BabelNet: The Automatic Construction, Evaluation and Application of a Wide-Coverage Multilingual Semantic Network. *Artificial Intelligence*, 193:217–250, 2012.

[87] Azlin Nordin, Nurul H. A. Zaidi, and Noor A. Mazlan. Measuring Software Requirements Specification Quality. *Journal of Telecommunication, Electronic and Computer Engineering*, 9(3–5):123–128, 2017.

[88] NSX bvba. NSX: Normalized Systems. https://normalizedsystems.org, 2020. Accessed: 2022-09-27.

[89] Dallin D. Oaks. *Structural Ambiguity in English*. Continuum International Publishing Group, London, 1st edition, 2010.

[90] About OMG. https://www.omg.org/about/index.htm. Accessed: 2019-01-19.

[91] OMG Unified Modeling Language (OMG UML). https://www.omg.org/spec/UML/2.5.1/PDF, 2017. Specification. Version 2.5.1. Accessed 2023-09-10.

[92] XML Metadata Interchange (XMI) Specification. https://www.omg.org/spec/XMI/2.5.1/PDF, 2015. Specification. Version 2.5.1. Accessed 2023-08-26.

[93] Harshvardhan J. Pandit, Declan O'Sullivan, and Dave Lewis. Using Ontology Design Patterns to Define SHACL Shapes. In *Proceedings of the 9th Workshop on Ontology Design and Patterns (WOP 2018) co-located with 17th International Semantic Web Conference (ISWC 2018)*, pages 67–71, 2018.

[94] Piano Chord Fingering – Which Fingers to Use When Playing Chords. https://www.piano-keyboard-guide.com/piano-chord-fingering-which-fingers-to-use-when-playing-chords, 2016. Accessed: 2021-01-09.

[95] Playable chords for string instruments? https://www.youngcomposers.com/t18240/playable-chords-for-string-instruments, 2009. Accessed: 2020-09-30.

[96] Anjalie Rajkumar, Mimi Taylor, and Nicholas Yap. Your bad requirements are costing you money. White paper. Deloitte Quality and Test Engineering, Deloitte, 2022. https://www2.deloitte.com/content/dam/Deloitte/uk/Documents/technology/deloitte-uk-your-bad-requirements-are-costing-you-money.pdf.

[97] Marcel Robeer, Garm Lucassen, Jan Martijn E. M. van der Werf, Fabiano Dalpiaz, and Sjaak Brinkkemper. Automated Extraction of Conceptual Models from User Stories via NLP. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 196–205, 2016. doi:10.1109/RE.2016.40.

[98] Danissa V. Rodriguez, Doris L. Carver, and Anas Mahmoud. An Efficient Wikipedia-Based Approach for Better Understanding of Natural Language Text Related to User Requirements. In *2018 IEEE Aerospace Conference*, pages 1–16, Big Sky, MT, 2018. IEEE Computer Society Press. URL: https://ieeexplore.ieee.org/document/8396645/, doi:10.1109/AERO.2018.8396645.

[99] Colette Rolland and Christophe Proix. A Natural Language Approach for Requirements Engineering. In Pericles Loucopoulos, editor, *Advanced Information Systems Engineering*, pages 257–277, Berlin, Heidelberg, 1992. Springer.

[100] Rudolf Rosa and Zdeněk Žabokrtský. Spacy, NLTK and other NLP frameworks. https://ufal.mff.cuni.cz/courses/npfl125#spacy_nltk_and_other_nlp_frameworks, 2022. Lecture notes, Introduction to Language Technologies (NLP125), Charles University. Accessed: 2023-09-06.

[101] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.

[102] Shinobu Saito, Mutsuki Takeuchi, Masatoshi Hiraoka, Tsuyoshi Kitani, and Mikio Aoyama. Requirements Clinic: Third Party Inspection Methodology and Practice for Improving the Quality of Software Requirements Specifications. In *2013 IEEE 21st International Requirements Engineering Conference (RE)*, pages 290–295, Los Alamitos, CA, USA, 2013. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/RE.2013.6636732, doi:10.1109/RE.2013.6636732.

[103] Wladimir Schamai, Philipp Helle, Nicolas Albarello, Lena Buffoni, and Peter Fritzson. Towards the Automation of Model-Based Design Verification. *INCOSE International Symposium*, 26(1):585–599, 2016. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2016.00180.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.2334-5837.2016.00180.x, doi:10.1002/j.2334-5837.2016.00180.x.

[104] Viliam Šimko, Petr Kroha, and Petr Hnětynka. Implemented Domain Model Generation. Technical Report No. D3S-TR-2013-03, Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University, Prague, 2013.

[105] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[106] Badariah Solemon, Shamsul Sahibuddin, and Abdul Azim Abd Ghani. Requirements Engineering Problems and Practices in Software Companies: An Industrial Survey. In Dominik Ślezak, Tai-hoon Kim, Akingbehin Kiumi, Tao Jiang, June Verner, and Silvia Abrahão, editors, *Advances in Software Engineering*, pages 70–77, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[107] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, United Kingdom, 9th edition, 2011.

[108] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[109] Riad Sonbol, Ghaida Rebdawi, and Nada Ghneim. Towards a Semantic Representation for Functional Software Requirements. In *2020 IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, pages 1–8, Los Alamitos, CA, USA, 2020. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/AIRE51212.2020.00007`, `doi:10.1109/AIRE51212.2020.00007`.

[110] George Spanoudakis and Anthony Finkelstein. A Semi-automatic Process of Identifying Overlaps and Inconsistencies between Requirements Specifications. In Collete Rolland and George Grosz, editors, *OOIS'98*, pages 405–424, London, 1998. Springer London.

[111] George Spanoudakis and Andrea Zisman. Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, pages 329–380. World Scientific, 2001.

[112] Robyn Speer and Julian Chaidez. API · commonsense/conceptnet5 Wiki · GitHub. `https://github.com/commonsense/conceptnet5/wiki/API#relatedness-of-a-particular-pair-of-terms`. Accessed: 2023-11-08.

[113] Robyn Speer, Joshua Chin, and Catherine Havasi. ConceptNet 5.5: An Open Multilingual Graph of General Knowledge. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 4444–4451. AAAI Press, 2017.

[114] Pantulkar Sravanathi and B. Srinivasu. Semantic Similarity between Sentences. *International research Journal of Engineering and Technology (IRJET)*, 4(1):156–161, 2017.

[115] Steffen Staab, Jens Lehmann, and Ruben Verborgh. Structured Knowledge on the Web 7.0. In *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, pages 885–886, Lyon, France, 2018. ACM Press. URL: https://dl.acm.org/citation.cfm?doid=3184558.3190666, doi:10.1145/3184558.3190666.

[116] Alistair Sutcliffe. Scenario-based Requirements Analysis. *Requirements Engineering*, 3(1):48–65, March 1998. doi:10.1007/BF02802920.

[117] Sparx Systems. Enterprise Architect User Guide: Import from XMI. https://sparxsystems.com/enterprise_architect_user_guide/16.1/model_exchange/importxmi.html. Accessed: 2023-11-27.

[118] Sparx Systems. UML modeling tools for Business, Software, Systems and Architecture. https://www.sparxsystems.com. Accessed: 2023-09-20.

[119] Akiyuki Takoshima and Mikio Aoyama. Assessing the Quality of Software Requirements Specifications for Automotive Software Systems. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 393–400, Los Alamitos, CA, USA, 2015. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/APSEC.2015.57, doi:10.1109/APSEC.2015.57.

[120] Pratvina Talele and Rashmi Phalnikar. Software Requirements Classification and Prioritisation Using Machine Learning. In Amit Joshi, Mahdi Khosravy, and Neeraj Gupta, editors, *Machine Learning for Predictive Analysis*, pages 257–267, Singapore, 2021. Springer Singapore.

[121] Ann Taylor, Mitchell Marcus, and Beatrice Santorini. *The Penn Treebank: An Overview*, pages 5–22. Springer Netherlands, Dordrecht, 2003. doi:10.1007/978-94-010-0201-1_1.

[122] Saurabh Tiwari, Deepti Ameta, and Asim Banerjee. An Approach to Identify Use Case Scenarios from Textual Requirements Specification. In *Proceedings of the 12th Innovations on Software Engineering Conference*, ISEC'19, pages 5:1–5:11, New York, NY, USA, 2019. ACM. doi:10.1145/3299771.3299774.

[123] Damiano Torre, Yvan Labiche, Marcela Genero, and Maged Elaasar. A Systematic Identification of Consistency Rules for UML Diagrams. *Journal of Systems and Software*, 144:121–142, 2018. URL: https://www.sciencedirect.com/science/article/pii/S0164121218301249, doi:10.1016/j.jss.2018.06.029.

[124] Universal Dependency Relations. https://universaldependencies.org/u/dep. Accessed: 2023-09-20.

[125] Universal POS tags. https://universaldependencies.org/u/pos. Accessed: 2023-09-20.

[126] Philip van Oosten, Dries Tanghe, and Véronique Hoste. Towards an Improved Methodology for Automated Readability Prediction. In Calzolari, Nicoletta and Choukri, Khalid and Maegaard, Bente and Mariani, Joseph and Odijk, Jan and Piperidis, Stelios and Rosner, Mike and Tapias, Daniel, editor, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pages 775–782. European Language Resources Association (ELRA), 2010. URL: http://www.lrec-conf.org/proceedings/lrec2010/pdf/286_Paper.pdf.

[127] Stefan Wagner, Daniel Méndez Fernández, Michael Felderer, Antonio Vetrò, Marcos Kalinowski, Roel Wieringa, Dietmar Pfahl, Tayana Conte, Marie-Therese Christiansson, Desmond Greer, Casper Lassenius, Tomi Männistö, Maleknaz Nayebi, Markku Oivo, Birgit Penzenstadler, Rafael Prikladnicki, Guenther Ruhe, André Schekelmann, Sagar Sen, Rodrigo Spínola, Ahmed Tuzcu, Jose Luis De La Vara, and Dietmar Winkler. Status Quo in Requirements Engineering: A Theory and a Global Family of Surveys. *ACM Transactions on Software Engineering and Methodology*, 28(2), 2019. doi:10.1145/3306607.

[128] Yue Wang, Irene L. Manotas Gutièrrez, Kristina Winbladh, and Hui Fang. Automatic Detection of Ambiguous Terminology for Software Requirements. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Elisabeth Métais, Farid Meziane, Mohamad Saraee, Vijayan Sugumaran, and Sunil Vadera, editors, *Natural Language Processing and Information Systems*, volume 7934, pages 25–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-38824-8_3.

[129] Wordnik Society, Inc. About Wordnik. https://www.wordnik.com/about. Accessed: 2023-11-08.

[130] Liping Zhao, Waad Alhoshan, Alessio Ferrari, and Keletso J. Letsholo. Classification of Natural Language Processing Techniques for Requirements Engineering, 2022. arXiv. arXiv:2204.04282.

[131] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J. Letsholo, Muideen A. Ajagbe, Erol-Valeriu Chioasca, and Riza T. Batista-Navarro. Natural Language Processing for Requirements Engineering: A Systematic Mapping Study. *ACM Comput. Surv.*, 54(3), 2021. doi:10.1145/3444689.

Reviewed Publications of the Author Relevant to the Thesis

[A.1] Šenkýř, D.; Kroha, P. Quality Measurement of Functional Requirements. In: *Proceedings of the 18th International Conference on Software Technologies*. SciTePress, Porto, 2023. doi:10.5220/0012148700003538.

[A.2] Šenkýř, D.; Suchánek M.; Kroha, P.; Mannaert, H.; Pergl, R. Expanding Normalized Systems from textual domain descriptions using TEMOS. In: *Journal of Intelligent Information Systems*. Springer, 2022. doi:10.1007/s10844-022-00706-8.

The paper has been cited in:

- Slifka, J.; Knaisl, V.; Pergl, R. Evolvable transformation of knowledge graphs into human-oriented formats. In: *Journal of Intelligent Information Systems*. Springer, 2023.
- Skotnica, M. Design of Systems Supporting Compliance Management. Dissertation thesis. Czech Technical University in Prague, 2023.
- Fernandes, W. J. de M. O estado da arte do processamento de linguagem natural em histórias de usuário. Bachelor thesis. Pontifícia Universidade Católica de Goiás, 2023.

[A.3] Šenkýř, D.; Kroha, P. Problem of Inconsistency and Default Consistency Rules. In: *New Trends in Intelligent Software Methodologies, Tools and Techniques*. IOS Press, Amsterdam, 2021. doi:10.3233/FAIA210063.

[A.4] Šenkýř, D.; Kroha, P. Problem of Inconsistency in Textual Requirements Specification. In: *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, Porto, 2021. doi:10.5220/0010421602130220.

The paper has been cited in:

- Talele, P.; Phalnikar, R. Multiple correlation based decision tree model for classification of software requirements. In: *International Journal of Computational Science and Engineering*. Inderscience Publishers, 2023.
- Vigneshwar, M. Using Neo4j DB system to store and query linguistic pattern. Master thesis. Czech Technical University in Prague, 2022.
- de Ribaupierre, H.; Cutting-Decelle, A. F.; Baumier, N.; Blumental, S. Automatic extraction of requirements expressed in industrial standards: A way towards machine readable standards? arXiv, 2021.

[A.5] Šenkýř, D.; Kroha, P. Problem of Semantic Enrichment of Sentences Used in Textual Requirements Specification. In: *Advanced Information Systems Engineering Workshops*. Springer, Cham, 2021. doi:10.1007/978-3-030-79022-6_7.

The paper has been cited in:

- Vigneshwar, M. Using Neo4j DB system to store and query linguistic pattern. Master thesis. Czech Technical University in Prague, 2022.

[A.6] Šenkýř, D.; Kroha, P. Patterns for Checking Incompleteness of Scenarios in Textual Requirements Specification. In: *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, Porto, 2020. doi:10.5220/0009344202890296.

[A.7] Šenkýř, D. SHACL Shapes Generation from Textual Documents. In: *Enterprise and Organizational Modeling and Simulation*. Springer, Cham, 2019. doi:10.1007/978-3-030-35646-0_9.

The paper has been cited in:

- Saleh, D. R. et al. On Generating SHACL Shapes from Collective Collection of Plant Trait Data. In: *Proceedings of the 2022 International Conference on Computer, Control, Informatics and Its Applications*. Association for Computing Machinery, New York, 2022.
- Álvarez, D. F. Extraction of structured semantic knowledge through data mining over social media. Doctoral thesis. University of Oviedo, 2023.
- Pareti, P.; Konstantinidis, G. A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs. In: *Reasoning Web. Declarative Artificial Intelligence*. Springer, Cham, 2022.

[A.8] Šenkýř, D.; Kroha, P. Problem of Incompleteness in Textual Requirements Specification. In: *New Knowledge in Information Systems and Technologies*. SciTePress, Porto, 2019. doi:10.5220/0007978003230330.

The paper has been cited in:

- Castillo-Motta, M.; Dorado-Cordoba, R.; Pardo-Calvache, C.; Orozco-Garces, C. Systematic Mapping of the Literature on Smells in Software Development Requirements. In: *Revista Facultad de Ingeniería*. 2023.
- Vigneshwar, M. Using Neo4j DB system to store and query linguistic pattern. Master thesis. Czech Technical University in Prague, 2022.
- Żeliński, J. Who is to blame for the project's failure? Purchaser! `https://it-consulting.pl/2020/12/24/kto-winien-porazki-projektu-zamawiajacy`, 2020. Last updated: 2022-09-12. Accessed: 2023-11-30.

[A.9] Šenkýř, D.; Kroha, P. Patterns of Ambiguity in Textual Requirements Specification. In: *New Knowledge in Information Systems and Technologies*. Springer, Cham, 2019. `doi:10.1007/978-3-030-16181-1_83`.

The paper has been cited in:

- Vigneshwar, M. Using Neo4j DB system to store and query linguistic pattern. Master thesis. Czech Technical University in Prague, 2022.

[A.10] Šenkýř, D.; Kroha, P. Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies*. SciTePress, Madeira, 2018. `doi:10.5220/0006827302310238`.

The paper has been cited in:

- Álvarez, D. F. Extraction of structured semantic knowledge through data mining over social media. Doctoral thesis. University of Oviedo, 2023.
- Suchánek, M. Towards a Normalized Systems Gateway Ontology for Conceptual Models. Dissertation thesis. Czech Technical University in Prague, 2023.
- Vigneshwar, M. Using Neo4j DB system to store and query linguistic pattern. Master thesis. Czech Technical University in Prague, 2022.
- Šimonová, S. Requirements Gathering for Specialized Information Systems in Public Administration. In: *2021 International Conference on Information and Digital Technologies (IDT)*. IEEE, 2021.

Remaining Publications of the Author Relevant to the Thesis

[A.11] Šenkýř, D.; Kroha, P. *Software Requirements Data Set.* Data set. Zenodo. doi:10.5281/zenodo.7897601, 2023.

[A.12] Šenkýř, D. *Processing, Checking, and Modelling of Textual Requirements Specications.* Doctoral minimum thesis, Faculty of Information Technology, Czech Technical University in Prague. Prague, Czech Republic, 2019.

[A.13] Šenkýř, D. *Patterns in Textual Requirements Specification.* Technical Report, No. TR-FIT-19-02, Faculty of Information Technology, Czech Technical University in Prague. Prague, Czech Republic, 2019.

[A.14] Šenkýř, D. *Generating of UML Entities from Textual Requirements Specification.* Master thesis, Faculty of Information Technology, Czech Technical University in Prague. Prague, Czech Republic, 2017.

Selected Relevant Supervised Thesis

[A.15] Šprysl, M. *Sentence Patterns Visualisation Tool.* Bachelor thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021.