Patterns in Textual Requirements Specification

David Šenkýř and Petr Kroha
Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Prague 6, Czech Republic

Keywords: Textual Requirements Specifications, Ambiguity, Inconsistency, Incompleteness, Glossary, Text Mining, Grammatical Inspection.

Abstract: In this paper, we investigate methods of grammatical inspection to identify patterns in textual requirements specification. Unfortunately, a text in natural language includes additionally many inaccuracies caused by ambiguity, inconsistency, and incompleteness. Our contribution is that using our patterns, we are able to extract the information from the text that is necessary to fix some of the problems mentioned above. We present our implemented tool *TEMOS* that is able to detect some inaccuracies in a text and to generate fragments of the UML class model from textual requirements specification. We use external on-line resources to complete the textual information of requirements.

1 INTRODUCTION

Writing requirements specifications in natural language is a common practice in big software houses. The market research (Luisa et al., 2004) states that nearly 80 % of all requirements specifications are written in natural language. The advantage of using natural language is that it can be interpreted both by the customer and by the analyst.

This increases quality of requirements that plays an important role in the whole development process, because the mistakes in requirements have the most expensive impacts (Landhäußer et al., 2014).

Formulation of requirements in natural language is also necessary because of the contract with clients. The contract is then the primary relevant source, which can be assessed in the event of a legal case.

Disadvantages of textual requirements are ambiguity, inconsistency, and incompleteness of the textual specification. Usually, 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 may have more meanings (ambiguity), text can obtain contradictions (inconsistency), and specifications of some features can be omitted (incompleteness).

A computerized processing of requirements formulated in natural language requires a collaboration of engineers and experts coming from computational linguistics. Nowadays, it is possible to choose from a variety of natural language processing systems. In our project, we used the system *CoreNLP* (Manning et al., 2014) as a component.

The human IT analyst is typically a domain expert for software construction, but he or she is typically not focused on the business or organization knowledge of the client. Therefore, his or her interpretation and understanding of textual requirements should be confronted with the knowledge of semantics of the text.

Computerized processing of natural language can be supported by acquired semantic knowledge from an appropriate corresponding ontology database. It may be difficult to obtain the ontology related to the client's domain – for many sectors, such ontology is not available. On the other hand, ontology databases for common language are available, e.g., WordNET, ConceptNet, DBPedia, Freebase, OpenCyc.

Our goal was to design and implement such a tool that assists mapping parts of textual requirements specification to corresponding fragments of static UML model with respect to possible ambiguity, inconsistency, and incompleteness of the textual specification.

Our paper is structured as follows. In Section 2, we discuss related works. We present the problems of textual requirements specifications in Section 3. Then we briefly explain the method of grammatical inspection in Sections 4. In Section 5, we discuss briefly the problems to be solved. Our contribution is presented in Sections 5.1. Our implementation is presented in Section 6, the case study is in Section 7. In Section 8, we conclude.

Šenkýř, D. and Kroha, P.
Patterns in Textual Requirements Specification.
DOI: 10.5220/0006827301970204
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 197-204
ISBN: 978-989-758-320-9

2 RELATED WORK

There are many interesting papers proposing that requirements engineering should be supported by tools based on the linguistic approach.

In paper (Rolland and Proix, 1992), the authors introduced a tool called OISCI. The mentioned tool processes French natural language. The approach presented in the paper targets the creation of the characterization of the parts of the sentence patterns that will be thereafter matched. OISCI also uses a text generation technique from the conceptual specification to natural language for the validation purposes.

In paper (Ambriola and Gervasi, 1997), a web-based system called *Circe* is presented that primary processes Italian natural language (but may be also adapted for other languages). It consists of partial tools. For our purposes, the most interesting tool is the main one called *Cico*. *Cico* performs recognition of natural language sentences and prepares inputs for other tools – graphical representation, metrication, and analysis. The paper presents the idea that requirements specification may be connected with a corresponding *glossary* describing all the domain-specific terms used in the requirements. The glossary also handles synonyms of terms. Similarly to the previous paper, *Cico* uses predefined patterns that are matched against the sentences from requirements specification.

In papers (Kof, 2005) and (Kof, 2004), the NLP approach is broken down into three groups. The first one is related to lexical methods, the second one build syntactical methods (part-of-speech tagging), and the last one is represented by semantic methods, i.e., by methods that interpret each sentence as a logical formula or looking for predefined patterns.

Linguistic assistant for Domain Analysis (*LIDA*) is a tool presented in the paper (Overmyer et al., 2001) from 2001. According to previous tools, *LIDA* is conceived as a supportive tool – it can recognize multi-word phrases, retrieves base form of words (stemming and lemmatization), presents frequency of words, etc. – but it doesn't contain algorithms for automatic recognition of model elements. The decisions about modeling are fully user-side, i.e., the user marks candidates for entities, attributes, and relations (inclusive operations and roles).

In paper (Arellano et al., 2015), there is presented tool *TextReq* based on *The Natural Language Toolkit* (NLTK). This toolkit is an open source platform for natural language processing in Python. It is an alternative to *Stanford CoreNLP* that is based on Java. From the papers mentioned above, the concept presented in this paper is the closest one to our approach.

There are also papers about generating the dynamic diagrams – e.g., *Activity* and *Sequence Diagram* (Gulia and Choudhury, 2016). A nice survey of papers on this field is given in (Dawood and Sahraoui, 2017).

3 PROBLEMS OF TEXTUAL REQUIREMENTS SPECIFICATIONS

The process of textual requirements processing, which is implemented in our tool TEMOS, is represented in Fig. 1. The schema also contains the swim lanes that visualize which parts are computed by our algorithms (*TEMOS* swim lanes), and which part is provided by *Standford Core NLP* framework (the middle swim lane). We recall that our tool accepted any free text as the input.

In the first phase, the text is perceived as a plain sequence of characters. We have to identify some cases that are not properly handled by *Stanford Core NLP* system, e.g., "his/her interpretation".

3.1 Nature Language Processing using Stanford CoreNLP

The text processing part of our tool is based on *Stanford CoreNLP* (Manning et al., 2014) and on its annotators. Annotators are procedures solving the different parts of the linguistic processing and generating notations that describe the results. The tokenization annotator parses the input text and provides lexical analysis. Tokens are used to build sentences by annotator *Words to Sentence Annotator*. The annotator *POS Tagger Annotator* provides the part of speech (POS) annotation (tagging) of every token – such as noun, verb, adjective, etc. Interpunction and other special characters are annotated with the same character that their represent. The *Morpha Annotator* generates base forms (lemmas) for every token.

From our point of view, the most interesting annotator is called *Dependency Parse Annotator*. It analyzes the grammatical structure of a sentence and looks for relationships between words (*nominal subject(s)* of verb, *dependency object(s)* of verb, etc.). Fig. 2 presents the output of the dependencies annotation. The dependency direction is indicated by an arrow.

Every sentence has one or more *root words*. These are the words that have no input dependencies. In Fig. 2, there is one root word – *bedroom*. We can see that the word *bedroom* is connected by *a compound dependency* to the word *hotel*. It may indicate that *a hotel bedroom* is a multi-word term, similarly like *a*

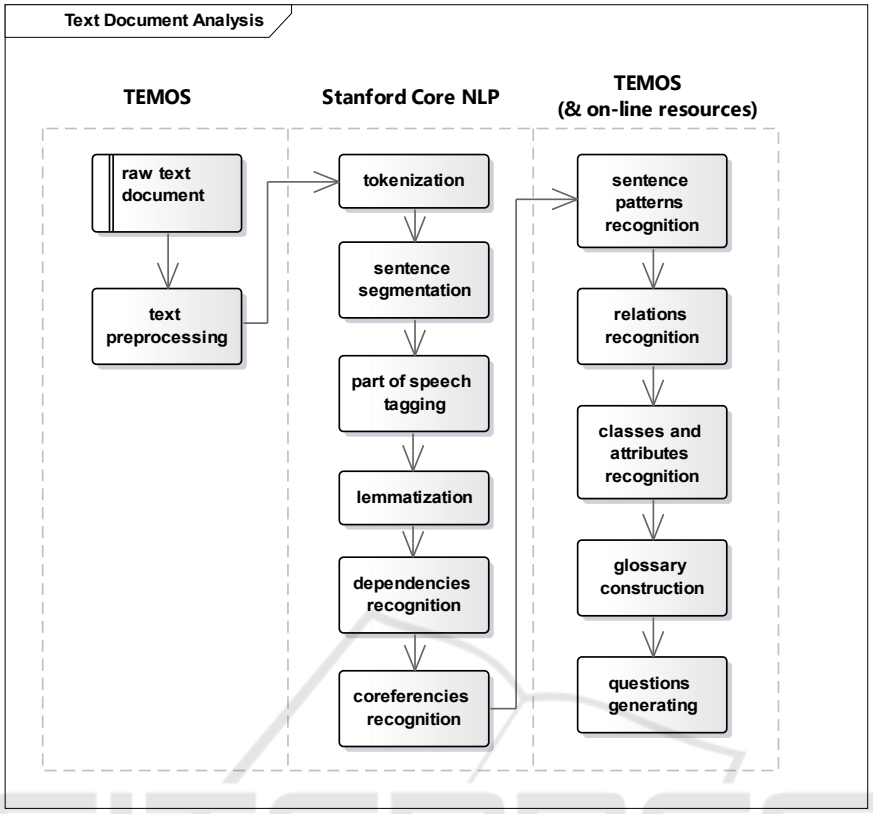

Figure 1: The Text Document Analysis.

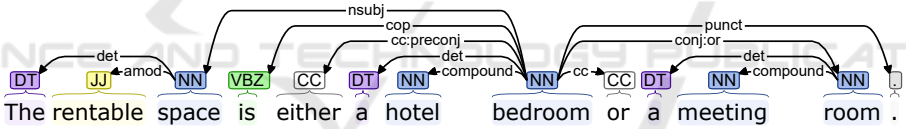

Figure 2: The Result of Dependency Parse Annotator – Enhanced++ Dependencies.

meeting room. The last annotator from *Stanford CoreNLP* that we use is *Coref Annotator*. Its purpose is to identify to which words pronouns refer – as shown in Fig. 3.

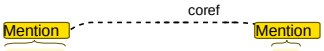

Figure 3: The Result of Coref Annotator.

4 GRAMMATICAL INSPECTION

The idea of mapping parts of the text document to *UML class diagram components* is based on the annotations. We use grammatical inspection – primarily based on the *dependencies recognition* and *part of speech tagging* – to identify grammatical roles of words in textual requirements, i.e., object, subject, etc. Similarly, as a user can highlight individual words in the text editor, he/she can also assign annotations to individual words or group of words using the editor included in *TEMOS*.

TEMOS introduces these annotation types:

- *Class Annotation* – a basic annotation that can exist separately.
- *Attribute Annotation* – the annotation that is associated with the owner of the *class annotation* type.
- *Relation Annotation* – the annotation that mediates a link between two and more *class annotations*. Therefore, *relation annotation* contains collections of *source class annotations* and *target class annotations*.

5 PROBLEMS TO BE SOLVED

Our approach opens a number of new problems that we discuss in this section.

5.1 Problem of Suitable Patterns

According to (Kof, 2004), our approach of classes and relations recognition belongs to *semantic methods*. Similarly to paper (Rolland and Proix, 1992) presented in Section 2, we adapt the pattern approach.

The pattern-based recognition is based on the idea that the grammatical role of a word in a sentence corresponds with the role of the entity assigned to the word that the entity plays in the model. The recognition process iterate through *root words*. With regard to the part of speech tag of the current *root word*, it is then matched against defined patterns to recognize a class, an attribute, or a relation. We use the following notation for the graphic representation of patterns:

- components with a *gray* background or foreground symbolize unrecognized parts of the pattern
- components with a *coloured* background or foreground symbolize recognized parts of the pattern,
- the notation *0..** above the word connection link means, like in the E-R schema, that a target word with this connection does not have to exists or such words may exist 1 or more at the same time,
- similarly, the notation *1..** allows more than 1 target word at the same time, but it also requires at least 1 such word.

5.1.1 Class/Attribute Sub-Pattern

The most of the following patterns contain this *class/attribute sub-pattern*. If there is a pattern indicating that some word may match the *class annotation* or *attribute annotation*, this sub-pattern will be used to find the expanding words of the base word to find the full expression.

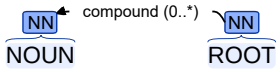

Figure 4: The Class/Attribute Sub-Pattern.

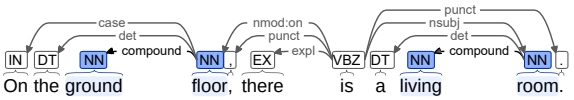

Figure 5: The Class/Attribute Sub-Pattern Example Matching.

5.1.2 Class-Specialization Pattern

The *class–specialization pattern* is defined by these rules:

1. The *root token* must be a *noun*. This root token will be the *class annotation* (C_1).
2. There must exist a *verb* (V) as a child of dependency of type *copula* (briefly *cop*). This verb must be *"to be"* verb.
3. There must exist a *noun* as a child of dependency of type *nominal subject* (briefly *nsubj*). This noun will be the *class annotation* (C_2).
4. If there exist any nouns as children of dependency of type *conjunct* (briefly *conj*), they will be the class annotations (C_3C_n).
5. The relation annotation is created with the verb V as a source token and C_1 as a source class annotation and C_3C_n as target class annotations.

Let's take a look at Fig. 2 again. The word *bedroom* as a root token meets this pattern. Therefore, *hotel bedroom* and *meeting room* are the specialization of *space*.

The matching of this pattern (Fig. 6) against Fig. 2 is illustrated in Fig. 7.

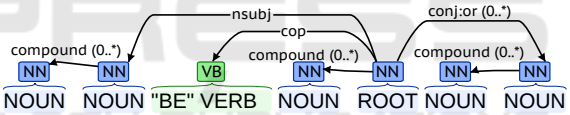

Figure 6: The Class Specialization Pattern.

5.1.3 Attribute Pattern #1

The goal of the attribute patterns is to identify attributes of classes and relations from textual requirements. If the root token is identified as a verb, then there are the following possibilities:

1. The *root token* must be *"to have"* verb, or *"to identify"* verb, or *"to contain"* verb.
2. There must exist a *noun* as a child of dependency of type *nominal subject* (briefly *nsubj*). This noun will be the *class annotation* (C_1).
3. There must exist a *noun* as a child of dependency of type *dependency object* (briefly *dobj*). This noun will be the *attribute annotation* (A_1).
4. If there exist any more nouns as children of dependency of type *dependency object*, they will be also marked by the attribute annotations (A_2A_n).

The matching of *"to have"* version of this pattern (Fig. 8) is illustrated in Fig. 9.

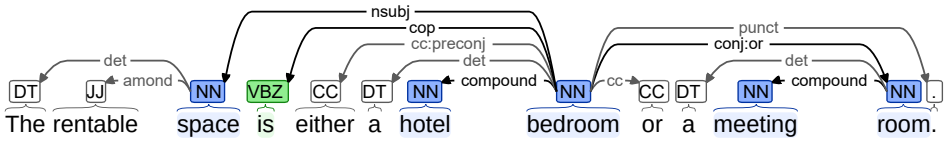

Figure 7: The Class Specialization Pattern Matching Example.

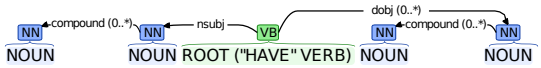

Figure 8: The Attribute Pattern #1.

5.1.4 Attribute Pattern #2

The following type of attribute pattern is identified if there is a noun that is not associated with any annotation and that meets the following conditions:

1. The *token* must be a *noun*. This noun will be the *attribute annotation* (A_1).
2. There must exist a *noun* as a child of dependency of type *nominal modifier of* (briefly *nmod:of*). This noun will have the *class annotation* (C_1).
3. If there exist any nouns as children of dependency of type *conjunction by and* (briefly *conj:and*), they will also have the attribute annotations ($A_2 A_n$).

The matching of this pattern (Fig. 11) is illustrated in Fig. 10.

5.1.5 General Relation Pattern

If no one of the above patterns can be matched, then the *general relation pattern* is applied. Its structure (*nominal subject(s)—verb—dependency object(s)*) is shown in Fig. 12.

5.1.6 Adverbial Clause Modifier Pattern

The previous pattern can be extended in the way of *adverbial clause modifier*. For example, in the sentence *"Each bank has its own central computer to maintain its own accounts and process transactions against them."*, the *general relation pattern* extracts information saying that the *bank* has the *central computer*, and the adverbial clause modifier pattern also identifies the purpose of this *computer* – maintaining *accounts* and processing *transactions*.

5.2 Problem of Glossary

The significant part of the textual requirements specification analysis is building *a glossary of terms*. Every class candidate is automatically introduced as a term of the glossary. Using the on-line ontology database *ConceptNet*[1], we present synonyms between terms and existing classes. The on-line English dictionary *Wordnik*[2] is used to provide a default definition of the glossary term.

5.3 Problems of Ambiguity, Inconsistency, and Incompleteness

The problem of ambiguity is related to our glossary - see above. Primary, searching for synonyms can identify the same entity that is presented under different labels in the text. Secondary, providing a default definition of every glossary term and asking for the user's approval can lead to a better specification of ambiguous entities via iterations.

The problem of inconsistency means that requirements specification can obtain some contradictions. Usually, it is not possible to reveal it by using only tools of text processing. In our previous work (Kroha et al., 2009), we used ontology modeling and description logic. However, this topic is very complex, and it is out of the scope of this paper.

The problem of incompleteness means that documents contain no proper or not complete description of entities. Similarly, an introduced entity which is not further used indicates missing information. At the very moment, our implemented tool uses two features to check incompleteness. We use transitivity of English verbs for checking incompleteness of the textual requirements and properties of the generated model entities to check the incompleteness of the model.

Transitivity of English verbs – relations are checked in the way of correct usage of the verb. English verbs can take 0, 1, or 2 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. On the contrary, the standalone sentence *"Administrator needs to maintain."* contains transitive verb *need*, and the information is missing about what needs to be maintained. Therefore, this sentence is suspicious and *TEMOS* generates warning for the user.

[1] http://www.conceptnet.io
[2] https://www.wordnik.com

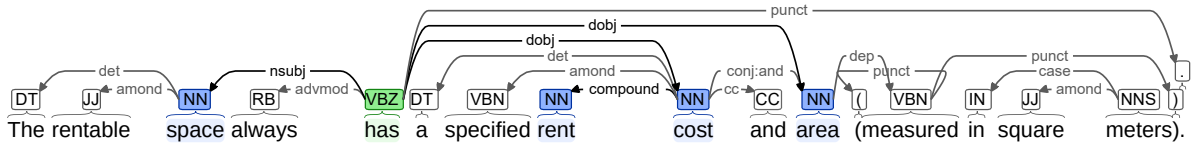

Figure 9: The Attribute Pattern #1 Matching Example.

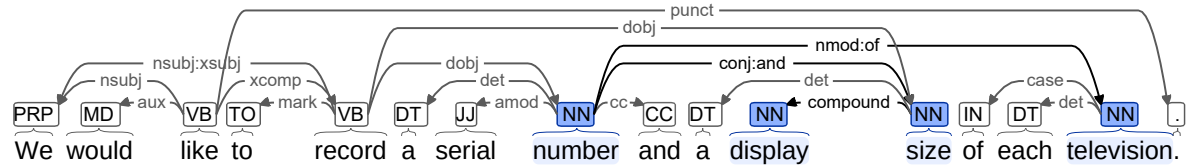

Figure 10: The Attribute Pattern #2 Matching Example.

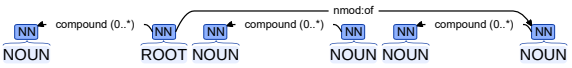

Figure 11: The Attribute Pattern #2.

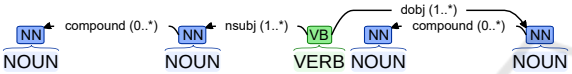

Figure 12: The General Relation Pattern.

Checking the model – we try to avoid the incompleteness by checking if a class has at least one attribute, and if it is in at least one relation to another class. We generate a warning, if it is not the case. This check is part of the *model validation* offered by *TEMOS*.

We use the detected problems as a source for generating warnings and questions for the user (Kroha, 2000). The user's answer will change the textual requirements specification, and the process makes a new iteration.

6 IMPLEMENTATION

Based on our approach, we designed and implemented a prototype of a software tool *TEMOS*. It is an acronym formed from **Te**xtual **Mo**delling **S**ystem. We would like to introduce a full-featured tool, therefore, in addition to the possibility of highlight parts of the textual requirements and their mapping to the *UML fragments* by the user, *TEMOS* will provide automatic text processing and automatic fragments mapping. Due to domain-specific requirements, *TEMOS* handles terms in an editable glossary. Based on the processed text, *TEMOS* generates the found models in various formats that can be used in the next step of processing (checking the model validity, generation of code fragments).

TEMOS was designed and implemented as *client-side multithreading application*. As a component, it includes *Stanford CoreNLP framework*, and therefore the primary functionality is available without having the Internet connection. If an Internet connection is available, *TEMOS* can use the on-line resources mentioned below. The disadvantage of using free on-line resources may be their limitation on the number of requests. TEMOS-architecture is based on the *Model-View-Controller* (*MVC*) pattern that corresponds to the *client-side JavaFX applications* architecture.

7 A CASE STUDY

Building models based on the textual requirement specification is a creative activity. Textual requirements are written by experienced analysts in cooperation with stakeholders using a specific simple structured language in form of well-formed sentences. Different analysts can create different models from the same requirements based on their experience. Therefore, testing the quality of generated models by our *TEMOS* tools and testing models, in general, is not an easy task and may be – from a certain point of view – subjective. In any case, it is out of the scope of this short paper.

We prepared an example with the following structure. First, there are original requirements with highlighted parts that were recognized by patterns mentioned in square brackets. These are followed by the generated model by *TEMOS* in the form of the *UML class diagram*. This diagram was acquired using *Enterprise Architect* after importing the model generated by *TEMOS* in the *XMI* format. The example is also supplemented with a brief comment on the quality of the generated model.

The mentioned patterns are indexed by this table.

1	general relation pattern
2	class/attribute sub-pattern
3	attribute pattern #1
4	attribute pattern #2
5	class-specialization pattern

7.1 Hotel Booking System

This example was created by us, and it mostly contains straightforward definitions (the structure of the sentence is in the format: *subject–verb–object(s)*) which are, of course, the best for automated processing.

7.1.1 Original Requirements

1. We would like to create a hotel booking system. *[0]*
2. Our **business group own**s many **hotel**s. *[1, 2]*
3. Every **hotel offer**s some rentable **spaces**. *[1]*
4. The rentable **space** always has a specified **rent cost** and **area** (measured in square meters). *[2, 3]*
5. The rentable **space** is either a **hotel bedroom** or a **conference room**. *[2, 5]*
6. The **hotel bedroom** has a unique **room number** and a number of **bed**s. *[3]*
7. The **hotel bedroom** may **contain** a **television**. *[1, 2]*
8. We would like to record a **serial number** and a **display size** of each **television**. *[2, 4]*
9. Every *hotel* **employ**s at least one *receptionist*. *[1]*
10. A **receptionist takes care of reservation**s from customers. *[1]*
11. Every **booking** has a **customer** and a selected rentable **space**. *[1]*
12. The **customer** is identified by **name**, **surname**, and **address**. *[3]*
13. The **reservation** contains a **start date** and an **end date**. *[2, 3]*
14. The **booking** also has a unique **identifier**. *[3]*
15. Every **conference room** has a **name** and a **maximum capacity**. *[2, 3]*
16. The **meeting room** can contain a **projection screen**. *[2, 3]*

7.1.2 The Generated Model

The generated model is in Fig. 13. The appearance of this *class diagram* corresponds to the status after accepting *class unification tips* (*booking=reservation* and *conference room=meeting room*).

7.1.3 Discussion

This example shows the recognition of all three types of annotations (*the class annotation*, *the relation annotation*, and *the attribute annotation*), and it also shows the case of *specialization* between the class *space* and subclasses *hotel bedroom* and *conference room*.

This result is already very close to the real processing by the analyst. A human analyst would probably not designate *the name* as a separate class – *TEMOS* did not consider *the name* to be an attribute because it is a shared information between two classes. The second attribute of the class *hotel bedroom* should be labeled as *number of beds* with respect to the 6th sentence – unfortunately, the current version of *Stanford CoreNLP* does not generate the necessary link here.

8 CONCLUSIONS

We designed and implemented the tool *TEMOS* that is able to handle the requirements for the software system written in plain text. Based on the tests from the Section 7, *TEMOS* can be used for generating drafts of *UML class models*. These models can be further modified or may be exported for further processing (*XMI* and *ECORE formats*) or may be visualized directly (*DOT format*).

Except of the suitable patterns, our contribution is also in using on-line resources to support resolving ambiguity and incompleteness.

Due to the high computational complexity (primarily due to *Stanford CoreNLP* analysis), the client-server architecture might be interesting. This architecture could also be used to collect data for the improvement of the analysis process.

Our further research is oriented on relation between syntactic ambiguity, i.e., the kind of ambiguity that can be resolved by means of computational linguistics (e.g., co-occurring of words, discussing contents of the glossary, etc.) with semantic ambiguity, i.e., the other kind of ambiguity that can be resolved only by means of roles of entities in the corresponding domain and problem models.

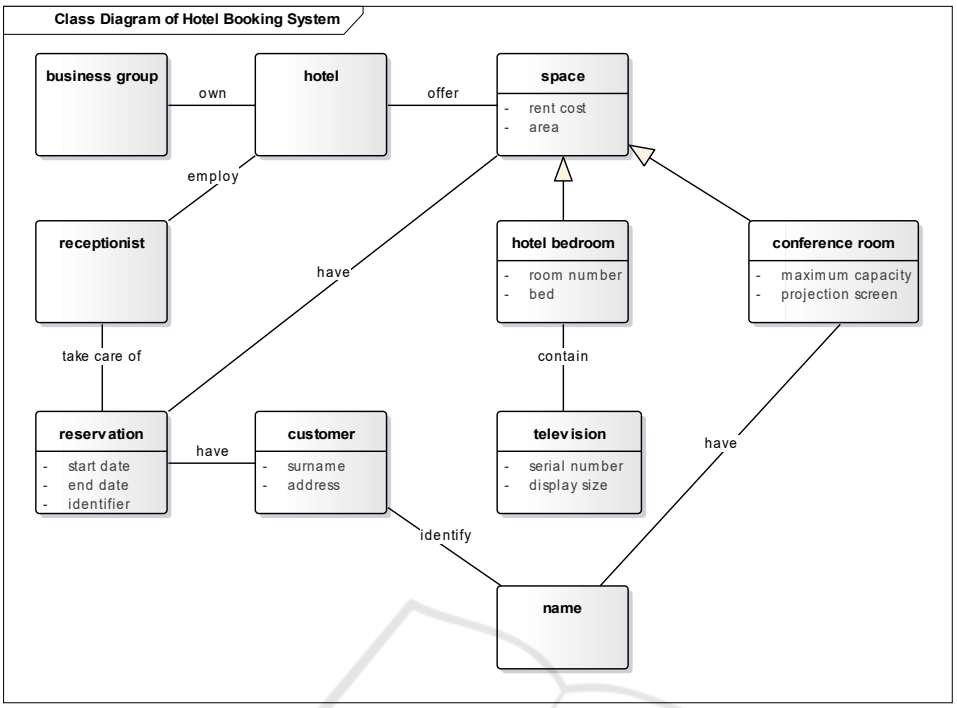

Figure 13: The Generated Class Diagram of Hotel Booking System.

REFERENCES

Ambriola, V. and Gervasi, V. (1997). Processing natural language requirements. In *Proceedings of the 12th International Conference on Automated Software Engineering (Formerly: KBSE)*, ASE '97, pages 36–, Washington, DC, USA. IEEE Computer Society.

Arellano, A., Zontek-Carney, E., and Austin, M. (2015). Frameworks for natural language processing of textual requirements. 8:230–240.

Dawood, O. and Sahraoui, A. (2017). From requirements engineering to uml using natural language processing survey study. *European Journal of Engineering Research and Science*, 2:44–50.

Gulia, S. and Choudhury, T. (2016). An efficient automated design to generate uml diagram from natural language specifications. In *2016 6th International Conference: Cloud System and Big Data Engineering (Confluence)*, pages 641–648.

Kof, L. (2004). An application of natural language processing to domain modelling: Two case studies. *International Journal on Computer Systems Science Engineering*, 20:37–52.

Kof, L. (2005). Natural language processing: Mature enough for requirements documents analysis? In *Natural Language Processing and Information Systems*, pages 91–102, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kroha, P. (2000). Preprocessing of requirements specification. In *Database and Expert Systems Applications*, pages 675–684, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kroha, P., Janetzko, R., and Labra, J. E. (2009). Ontologies in checking for inconsistency of requirements specification. *Third International Conference on Advances in Semantic Processing*, pages 32–37.

Landhäußer, M., Körner, S. J., and Tichy, W. F. (2014). From requirements to uml models and back: How automatic processing of text can support requirements engineering. *Software Quality Journal*, 22(1):121–149.

Luisa, M., Mariangela, F., and Pierluigi, N. I. (2004). Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60.

Overmyer, S. P., Lavoie, B., and Rambow, O. (2001). Conceptual modeling through linguistic analysis using lida. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 401–410, Washington, DC, USA. IEEE Computer Society.

Rolland, C. and Proix, C. (1992). A natural language approach for requirements engineering. In *Advanced Information Systems Engineering*, pages 257–277, Berlin, Heidelberg. Springer Berlin Heidelberg.