Patterns for Checking Incompleteness of Scenarios in Textual Requirements Specification

David Šenkýř[a] and Petr Kroha[b]
Faculty of Information Technology, Czech Technical University in Prague, Czech Republic

Keywords: Requirements Specification, Text Processing, Scenarios, Grammatical Inspection, Incompleteness, Domain Model.

Abstract: In this contribution, we investigate the incompleteness problem in textual requirements specifications. Missing alternative scenarios are one of the incompleteness sources, i.e., descriptions of processing in the cases when something runs in another way as expected. We check the text of requirements specification using linguistic patterns, and we try to reveal scenarios and alternative scenarios. After that process is finished, we decide whether the set of alternative scenarios is complete. As a result, we generate warning messages. We illustrate our approach with examples.

1 INTRODUCTION

Very often, textual requirements specification does not contain complete information about the system to be constructed because it is difficult to obtain all required information before software design and implementation starts. Many years of practice have shown that using incomplete information in requirements specification leads to incomplete or wrong models. It can cause customers to reject the specified system because it does not meet all their expectations. It also means that costs and schedules have been underestimated. The resulting program has to be laboriously enhanced and adapted. A study (Standish, 1994) implicated that incomplete requirements caused 12.3 % of costs overrun and 13.1 % of failed projects.

The problem is getting more complicated by the fact that requirements specification are engineered in incremental, iterative, parallel (more teams) manner, i.e., they are changed and amended during the software development process under schedule with milestones and a limited budget. This fact may have an impact on their incompleteness and inconsistency.

Requirements specification follows the requirements analysis, which is typically based on various requirements models. In (Firesmith, 2005), these models are listed, starting with context model, data model, decision model, event model, performance model, process model, safety and security model, state model, and use case model. In our approach, we focus only on scenarios known from functional requirements based on a process model. Scenarios and use cases have very much in common. Scenarios are often used to make use cases more specific. This means that our view on requirements is a dynamic view, especially a scenario view, which is based on sequence diagrams and communication diagrams.

There are more reasons for incomplete requirements specification.

First, the software system requirements specification describes only a specific part of the real system to be modelled – a simplified model omitting some details. We are interested only in a subset of all existing objects, properties, and relationships of the real system. For example, when writing information systems, this subset is given by supposed queries, use cases, and scenarios.

Second, subject matter experts working with the analyst on the textual version of requirements often take certain information for granted and suppose that some facts are apparent, and they do not mention them. However, what is evident for a user of a protein information system is usually not evident for the software analyst.

Third, some details or some scenarios are forgotten at the very beginning of the project, or some queries are appended later without to worry whether the underlying model contains the necessary objects, properties, or relationships that are necessary to

[a] https://orcid.org/0000-0002-7522-3722
[b] https://orcid.org/0000-0002-1658-3736

Šenkýř, D. and Kroha, P.
Patterns for Checking Incompleteness of Scenarios in Textual Requirements Specification.
DOI: 10.5220/0009344202890296
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 289-296
ISBN: 978-989-758-421-3

produce the information in demand.

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. We investigated this problem in our previous paper (Šenkýř and Kroha, 2019). The second scope includes the whole document of the textual requirements specifications, i.e., all its sentences. This problem is the topic of our paper, in which we describe how to reduce this type of incompleteness.

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

Missing alternative scenarios can cause users to be disappointed and frustrated with the system, and it can cause customers to reject the system.

In our approach, we check the text of requirements specification using linguistic patterns to reveal scenarios and alternative scenarios. However, the semantic relations in real systems are often so complex and hidden that they cannot be solved automatically. Thus, our implemented tool TEMOS generates a warning message in the case when it finds a suspicious formulation.

Our paper is structured as follows. In Section 2, we discuss related work. Our approach is presented in Section 3. The implementation, used data and experiments are described in Sections 5 and 6. Finally, in Section 7, we conclude.

2 RELATED WORK

Completeness of requirements specification is a complex problem discussed since years in details in many publications, e.g., in (Firesmith, 2005).

Our approach is based on scenarios. In (Sutcliffe, 1998), 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 (Ferrari et al., 2014). 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 backwards functional completeness, that the paper (Ferrari et al., 2014) 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 (Šenkýř and Kroha, 2018)) and check in the sense of the forward functional completeness (Šenkýř and Kroha, 2019). 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 in our approach. In contrary to the approach in (Ferrari et al., 2014), 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 (Li, 2015), 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 (Eckhardt et al., 2016), sentence patterns are used to uncover incompleteness with performance requirements. According to the unified model, the performance requirements describe time behaviour, throughput capacity, and cross-cutting. The sentence patterns used in the paper are completely different from our sentence patterns.

In (Dalpiaz et al., 2018), the authors explore potential ambiguity and incompleteness based on the terminology used in different viewpoints. They combine possibilities of NLP technology with information visualization. Their approach is completely different from our approach.

The approach based on NLP techniques and grammatical inspection methods to extract use case scenarios is proposed in (Tiwari et al., 2019). It is a similar approach to ours, but the question of incompleteness is not discussed there.

In (Bäumer and Geierhos, 2018), the authors developed methods of detecting quality violations in a requirements specification called *linguistic triggers*. The authors combine their approach to the problem of incompleteness with the problem of ambiguity. To detect incompleteness, the authors use their approach published in (Bäumer and Geierhos, 2016), in which their linguistic triggers work in two steps. In the first step, the detection via *predicate argument analysis* is used to assign semantic roles, e.g., *agent*, *theme*, *beneficiary*, to the recognized predicate. Some verbs have "rich predicates", e.g., the verb "send" is a three-place predicate because it requires the agent (sender), the theme (send), and the beneficiary argument (sent-to). The second step is compensation. Using *similarity search component* known from *information retrieval* (IR) domain, they try to find the potentially missing part *sent-to* based on software descriptions gathered from one software-to-download portal. In our papers, we discuss the problems of ambiguity and incompleteness separately.

3 OUR APPROACH TO THE PROBLEM OF INCOMPLETENESS USING SCENARIOS

The goal of our method described in this contribution 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 chapters.

In this paper, we present some simple examples that concern user interface. As the norms ISO/IEC/IEEE 29148/2018 and IEEE Std 1012:2016 explain, specification of a user interface is part of requirements specification.

3.1 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:

- description of the system context (input events, system output, i.e., system's communication with the actors out of the system),
- description of system usage including user's goals (including use cases) and system function,
- 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 requirements specification.

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

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

(Example 1). Assume, we have textual requirements specification of a text editor. One of use cases is described as a functional requirement.
Normal case scenario: To edit a text file, the user has to click the button OPEN, to choose the file he/she wants to edit. After the user see the file content, he/she provides the changes of its textual content. To save the changes, the user uses button SAVE.
Alternative (exceptional) scenarios:

Scenario 1: ... If the user has got the message "The file cannot be found" the user has to do the following ...

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 ...

(End of Example 1)

To have the possibility of testing the existence of these alternative scenarios, we need a class *File* containing attributes *File-Status* (values: *Exists*, *Does not exist*, *Opened*, *Closed*) and *File-Type* (values: *docx*, *doc*, *rft*, *txt*, *tex*, etc.) in the model. Some text editors can open files of types *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 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 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.

3.2 The Algorithm

The algorithm implemented in our tool has to follow these steps:

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 3.2.1 discusses the model construction in more detail.
3. To find sets of values of each attribute as described in Section 3.2.2.
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 3. This core step of the whole algorithm is described separately in Section 4.
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.

3.2.1 Static UML Model Construction

Using grammatical inspection, our tool TEMOS finds classes, their attributes and constraints involved, as we described in our paper (Šenkýř and Kroha, 2018). After we have classes and their attributes, our tool is looking for their values, and it builds a corresponding set of values to each attribute of each class that participates in *the normal case scenario*.

These sets need not to be built only from the text of requirement specifications. Using some predefined knowledge databases, we can generate warning messages, e.g., we find in a knowledge database that a file (from Example 1) can have a File-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.

3.2.2 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 $Card(S\text{-sets})$.

In 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 $DB_{C,Attr}$, and each of these sets has its cardinality $Card_{S,C,Attr}$.

In the first step, we can call it calibration, cardinalities of sets constructed from requirements will be compared with 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.

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 2). In the traffic simulation, the class representing *Traffic Light* has the corresponding set of attributes such as {Type, Year of Production, Light, etc.}. From the specification, we know possible values of attribute *Light*. It’s this set: {Red, Orange, Green}.

Scenario for Light equals Red:
Stop procedure is applied...
Scenario for Light equals Green:
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?”

(End of Example 2)

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 separated by commas, or the multiple values are hidden behind the noun “others”.

4 PATTERNS TO FIND SCENARIOS

To implement the method and the algorithm (Section 3.2) described above, we replace the problem of finding scenarios by the problem of finding typical textual patterns that indicate the presence of scenarios in a text of requirements specification. We reuse the already presented (Šenkýř and Kroha, 2018) idea of a *grammatical inspection* and *sentence patterns* following (Rolland and Proix, 1992). So, we construct the corresponding patterns, we apply them on 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 cover situation where the class name and names of its attributes have to be mentioned. The patterns from the second group cover situation where the values of some attribute are used but the attribute name is not explicitly mentioned.

4.1 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 context situation is:
Class File has methods Open (means Open existing file), Enter data (or Import file from an external device), Save (without changing Name, Directory and Type), and Save As (Changes possible).

Normal Case Scenario:
To process an existing file (CLASS NAME) that can be seen in the window of the file manager, the user uses Open item in the File menu.

Alternative Scenario:
To process an existing file (CLASS NAME) that cannot be seen in the window of the file manager, the user uses 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.

4.1.1 Values Conditional Pattern

This category covers a situation where the requirements enumerate behaviour 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 are concerned.

Figure 1: Values Conditional Pattern #1.

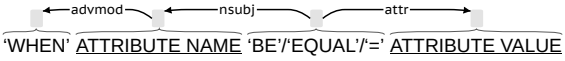

Figure 2: Values Conditional Pattern #2.

4.1.2 Incomplete Conditional Pattern

Typically, the requirement describes the prospective situation in a conditional way (e.g., if something is (successfully) loaded/processed/OK), but there are missing alternatives. Based on this observation, we can create two sets. The first one contains prospective nouns (Fig. 3) and verbs (in the “-ed” form, Fig. 4) and the second one contains their opposites. When the scenario mapping prospective situation appears, it’s time to check if the opposite exists.

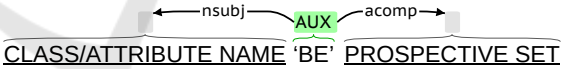

Figure 3: Incomplete Conditional Pattern #1.

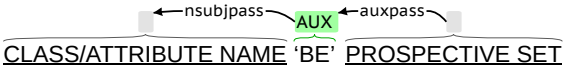

Figure 4: Incomplete Conditional Pattern #2.

4.2 Patterns Covering Values of Attribute

Other situations not handled by the previous category are presented here. Let’s focus on sentences where the name of the attribute is not mentioned. And let’s focus on properties of attributes such as uniqueness.

4.2.1 Values in Enumeration Pattern

This pattern considers enumerations, and it’s based on the text structure. Different from *Values Conditional*

Pattern, we do not require a conditional form in this case.

As shown in Example 3[1], all (three) points of enumeration listed in the part of *Installation* together cover all values of one specific attribute (*Operating system*). But only two of them are discussed in the part of *Configuration*.

(Example 3).
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 available the application from https://tunnelblick.net that contains both Open VPN and the graphical interface.

5.13.2 Configuration of OpenVPN

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

(End of Example 3)

You 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 illustrating Fig. 5.

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.

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

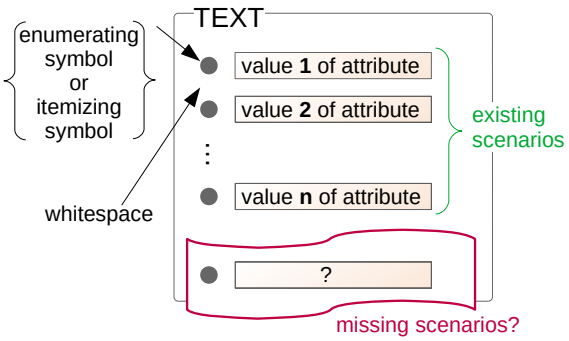

Figure 5: Values in Enumeration Approach.

4.2.2 Unique Attribute Pattern

In the requirements, some attributes may be marked as unique. When such a unique attribute is used within the extracted positive use case scenario, there should surely be the alternative one. We can find it via pattern in Fig. 6 where the *unique set* should consists of words such as *taken*, *occupied*, *used*, etc. Attribute representing "username" is a typical example.

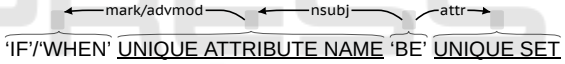

Figure 6: Unique Attribute Pattern.

5 IMPLEMENTATION

The current version of our tool TEMOS is written in Python, and it is powered by *spaCy*[2] NLP framework in version 2.2 (we use the features such as *tokenization*, *sentence segmentation*, *part-of-speech tagging*, *lemmatization*, *dependency recognition*, *co-reference recognition*). We use texts written in English, and for this purpose, we use the pre-trained model called *en_core_web_lg* (available together with spaCy installation).

The workflow of our implemented tool extends the original one presented in (Šenkýř and Kroha, 2018). After base text classification (provided by spaCy) and before we start to generate the output model, it makes sense to check the text against inaccuracies. So far, we tackle the ambiguity and incompleteness issues. This check is optional but recommended. Based on the methods presented in this paper, we extended the module denoted to incompleteness revealing. In Fig. 7, there is a screenshot of TEMOS application.

[2]https://spacy.io

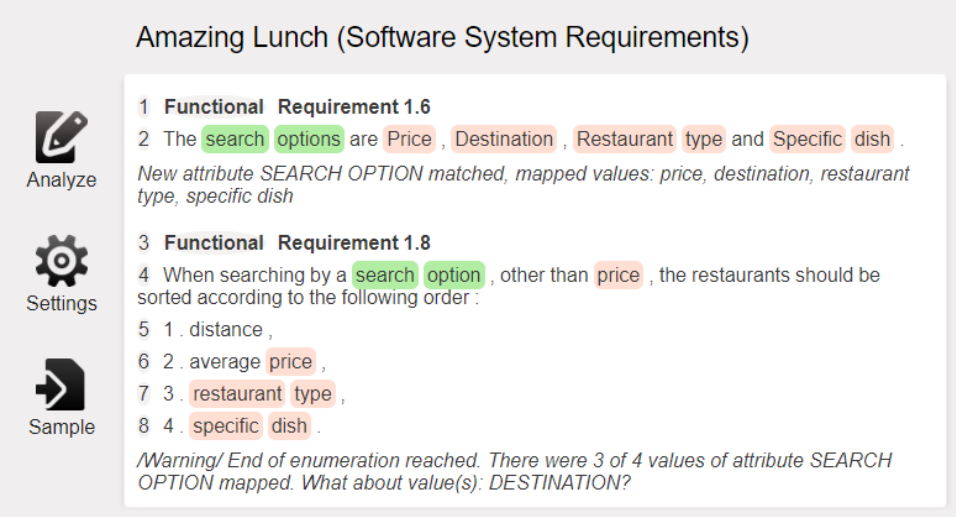

Figure 7: TEMOS Application.

6 DATA AND EXPERIMENTS

The method we described is not perfect. We are not able to find all the forgotten alternative scenarios. It can happen that there are more alternative scenarios concerning a class attribute and its values, as we can show in the following example.

(Example 4). From textual requirements specification of a library information system, we find that its model contains a class Reader having attribute Date-of-Birth. There is a functional requirement describing how readers borrow books.

Functional requirement: A reader lays the book on the screen of the indicating machine, insert his/her membership card in the slot of the machine, and pushes the button READY.

Normal case scenario: After a green light starts to blink, the process is finished, and the following message occurs: "The process of borrowing a book has been finished successfully."

Alternative scenario: After a red light starts to blink, the process is finished by an error, and the following message occurs: "Readers younger than 18 years cannot borrow books for adults."

We have our alternative scenario concerning the Reader-Age and its value. Of course, there may exist a set of other alternative scenarios concerning the Reader-Age and its value, e.g., "Reader older than 70 years can borrow books of the library free of charge."

It should be mentioned in the functional requirement: Every reader will be charged by 50 Euro at the January, 1st.

Both Age-restrictions are described in Library rules that had to be elicited before requirements specification has been formulated.

The only action we could do is to generate a question whether there are some additional possibilities depending on the reader's age.

(End of Example 4)

We continue with an example from published software system requirements where our methods could help. In Fig. 7, there is a screenshot of processing the next example by our tool.

(Example 5). In (Geagea et al., 2010), there is *Functional Requirement 1.6* stating "The search options are Price, Destination, Restaurant type and Specific dish". Our tool map these four options as values of the attribute *search option*.

Functional Requirement 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 attribute *search option* are mentioned. The pattern *5.2.1 Values in Enumeration Pattern* was applied. Therefore, our tool generates a warning regarding the missing value *destination*. The analyst based on this warning can decide whether the distance is an expression of destination in this context, and if the meaning is clear.

(End of Example 5)

---

7 CONCLUSIONS

The identification of incompleteness in requirements specifications is still an open issue in requirements engineering.

We presented our method that reveals scenarios and alternative scenarios in textual requirements specification using patterns. In the first phase, we construct a UML model as we described in our previous paper (Šenkýř and Kroha, 2018). Using the model, we find values of attributes of classes that take parts in scenarios and alternative scenarios. Then we compare sets of attributes mentioned in alternative scenarios with sets of attributes from the model and decide about this kind of requirements specification completeness.

Unfortunately, it is difficult to get large textual requirements specifications from software companies. Usually, these documents are classified as confidential. Software developers do not agree with the publishing. This is the reason why we cannot present statistical data supporting our patterns.

In the future, we will test the possible existence of some more complicated textual formulations of scenarios and their correspondence with a domain model.

ACKNOWLEDGEMENT

This research was supported by the grant of Czech Technical University in Prague No. SGS17/211/OHK3/3T/18.

REFERENCES

Bäumer, F. S. and Geierhos, M. (2016). Running Out of Words: How Similar User Stories Can Help to Elaborate Individual Natural Language Requirement Descriptions. In Dregvaite, G. and Damasevicius, R., editors, *Information and Software Technologies*, volume 639, pages 549–558. Springer International Publishing, Cham.

Bäumer, F. S. and Geierhos, M. (2018). 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.

Dalpiaz, F., van der Schalk, I., and Lucassen, G. (2018). Pinpointing Ambiguity and Incompleteness in Requirements Engineering via Information Visualization and NLP. In Kamsties, E., Horkoff, J., and Dalpiaz, F., editors, *Requirements Engineering: Foundation for Software Quality*, pages 119–135, Cham. Springer International Publishing.

Eckhardt, J., Vogelsang, A., Femmer, H., and Mager, P. (2016). Challenging Incompleteness of Performance Requirements by Sentence Patterns. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 46–55, Beijing, China. IEEE Computer Society Press.

Ferrari, A., dell'Orletta, F., Spagnolo, G. O., and Gnesi, S. (2014). Measuring and Improving the Completeness of Natural Language Requirements. In Salinesi, C. and van de Weerd, I., editors, *Requirements Engineering: Foundation for Software Quality*, pages 23–38, Cham. Springer International Publishing.

Firesmith, D. (2005). Are Your Requirements Complete? *Journal of Object Technology*, 4(1):27–43.

Geagea, S., Zhang, S., Sahlin, N., Hasibi, F., Hameed, F., Rafiyan, E., and Ekberg, M. (2010). Software Requirements Specification: Amazing Lunch Indicator. Available from: http://www.cse.chalmers.se/~feldt/courses/reqeng/examples/srs_example_2010_group2.pdf.

Li, A. (2015). Analysis of Requirements Incompleteness Using Metamodel Specification. Master's thesis, University of Tampere.

Rolland, C. and Proix, C. (1992). A Natural Language Approach for Requirements Engineering. In *Advanced Information Systems Engineering*, pages 257–277, Berlin, Heidelberg. Springer.

Šenkýř, D. and Kroha, P. (2018). Patterns in Textual Requirements Specification. In *Proceedings of the 13th International Conference on Software Technologies*, pages 197–204, Porto, Portugal. SCITEPRESS – Science and Technology Publications.

Šenkýř, D. and Kroha, P. (2019). Problem of Incompleteness in Textual Requirements Specification. In *Proceedings of the 14th International Conference on Software Technologies*, volume 1, pages 323–330, Porto, Portugal. INSTICC, SCITEPRESS – Science and Technology Publications.

Standish (1994). The CHAOS Report (1994). Technical report, The Standish Group.

Sutcliffe, A. (1998). Scenario-based Requirements Analysis. *Requirements Engineering*, 3(1):48–65.

Tiwari, S., Ameta, D., and Banerjee, A. (2019). 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. ACM.