Improving Quality of Software Requirements by Using a Triplet Structure

Bruel Gérançon, Sylvie Trudel
Department of Computer Science
Université du Québec à Montréal (UQAM)
Montréal, Canada
e-mail: gerancon.bruel@courrier.uqam.ca, trudel.s@uqam.ca

Abstract— The development of a quality software system is a priority in the domain of software engineering. Quality requires to define unambiguous and consistent requirements that are not conducive to various interpretations. Indeed, the success of the realization of a software system depends largely on the phase of software requirements specification. The requirements specification phase consists, among other things, in describing in a precise and unambiguous manner the characteristics of the system to be developed. Moreover, the techniques for writing software specification documents used in the industry don't facilitate to define unambiguous and consistent requirements. In industry, software requirements are often written in natural language, and no technical details are specified. Thus, software requirements are incomplete, inconsistent and prone to ambiguities, and therefore interpretation errors can easily be made by analysts. This article introduces a new technique for writing and developing software requirements that could help to reduce the ambiguities and inconsistencies in the document of specification software requirements. Our technique is validated by the development of a new tool that detects the ambiguities and inconsistencies in the software requirements, and generates the potential methods and classes from requirements written in natural language. Our tool integrates a set of techniques in natural language processing (NLP), and in artificial intelligence which helps to improve the software requirements quality.

Keywords; Software requirements quality, Triplet, Natural language processing (NLP), Artificial intelligence.

I. INTRODUCTION

Requirements engineering is an important phase of the software development life cycle. By definition, a software requirement is a condition that a software or system must be able to meet. In other words, a software requirement is a capability that a system must exhibit to satisfy a contract between a customer and a supplier. In indeed, the software requirements quality has a significant impact on the quality of all phases of the software development process. According to Boehm and al. [3] and Laporte [4], nearly 50% of software defects are from the software requirements specification. Moreover, any injected defect requires a higher cost to fix in other phases of the software life cycle. For example, a defect detected in requirement specification costs about 1$ of fixing [3]. While, the cost of fixing for this same defect detected after deployment is estimated at 100$ [3]. Moreover, requirements are written in natural language, they are often incomplete, unclear and prone to ambiguities [6][22].

II. SOFTWARE REQUIREMENTS SPECIFICATION LANGUAGES

Software requirements specification languages could be mainly classified in two (2) categories, which are respectively formal and natural languages [11]. The formal specification languages refer to requirement defined to be as unambiguous as possible, in using syntaxes. While the non-formal specification languages refer to software requirements described in natural language.

A. Formal language

Formal languages are subject to a well-defined syntax and semantics. Each formulation is subject to an unique interpretation which makes it possible to define and explain the requirements in a precise and unambiguous manner [11]. The big advantage of formalizing specifications is that it allows an interpretable representation by a machine [11]. However, transforming non-formal to formal specifications requires human expertise and experience, since the complexity of formal language syntax is not understandable by non-experts [8][11]. In addition, the specification of software requirements is a collaborative task between the customer (product owner) and the analysts. These requirements must be specified and expressed in a common language, simple and understandable for all stakeholders, experts and non-experts. Therefore, software requirements should be specified in neutral technical language that facilitates understanding of all stakeholders [6].

B. Natural language

Software requirements are generally specified and written in natural language in order to allow a wider understanding between the different stakeholders: customers, analysts, designers and developers [8] [12]. Due to its expressive aspect, the specification of software requirements in natural language (LN) are subject to various interpretations and could contain inconsistent information. Therefore, the specifications in LN are not directly interpretable by a machine. Developing software requirements of quality could depend on the technique used for writing them. For example, Jacobson et al. [12] propose the technique of "Use Cases" to write software requirements. Beck and West [13], for their

979-8-3503-2028-2/22/$31.00 ©2022 IEEE
DOI 10.1109/CSCI58124.2022.00358

part, propose "User Stories" as a technique for writing software requirements. These techniques are texts widely used to identify and record the functional requirements of the software. Other authors propose the Acceptance Test Driven Development (ATDD)method, a method associated more particularly with Extreme Programming (XP). In this article, we will present the limitations of software requirements writing techniques such as Use Cases and User Stories. Afterward, we will introduce a new technique for writing software requirements which helps to reduce the ambiguities in requirements and improve the quality software requirements.

III. LIMITATIONS OF SOFTWARE REQUIREMENT WRITING TECHNIQUES

The industry-adopted software requirement writing techniques have several limitations. Indeed, one of the limits of the classic or traditional approach to writing software requirements comes from the fact that the techniques, in particular Use Case and Use Stories, used to specify and describe software requirements are in natural language, with unnecessary details. Several defects are injected in the software requirements specification phase [3][9]. According to Ambler [23], these techniques increase the risk of failure of software development projects, since they describe a large percentage of software specifications that are never implemented. Additionally, the classical approach to requirements writing fails to solve the problem of requirement semantics, since natural language is inherently ambiguous. Software requirements are often incomplete, inconsistent and prone to ambiguities, and therefore misinterpretations can easily be made by software engineers [6][22]. Ambiguity and inconsistency in requirements are central problems with the current approach to requirements engineering. Additionally, since the traditional approach requires writing detailed requirements early in a project life cycle and most of these requirements don't often reflect the unique needs of users. Therefore, it increases the risk of failure to produce software that meets the customer's business value. Moreover, the limit of the classic approach could lie in the traceability of requirements throughout all phases of the software life cycle (analysis, design, coding, testing). It is often observed during the testing phase that the final software does not correspond to the software requirements specified in analysis and design [16][18]. As a result, wouldn't it be necessary to find new methods, techniques or approaches for writing software requirements?

In addition to these limitations, the software requirements are complex, inconsistent, not concise, and subject to various interpretations [6][14][22]. These factors influence the reading, understanding, interpretation and management of the requirements. Could the using of a triplet structure in requirements engineering resolve these limitations and reduce the ambiguity of software requirements? Finally, according to Nuseibeh and Easterbrook [16], the limitations of the classical approach to requirements writing lie primarily in the non-specification of semantic links between development artefacts during the early stages of development. Software requirements are potentially inconsistent, as they are described by stakeholders each with different perspectives, conceptual differences and vested interests. Second, the limitation comes from the difficulty for stakeholders to produce a formal specification of the requirements [16]. Third, the limitation of the classic approach to specifying and writing software requirements is that the traceability links between requirements and development artifacts are rarely specified.

IV. TRIPLET STRUCTURE

The proposed triplet structure is a model that allows to define and represent the software requirements as a triplet, in order to reduce ambiguities and inconsistencies in the description of software specifications [1][2] [10]. The goal is to define the software requirements in a simple, concise and clear as a triplet model consisting of a subject, a predicate, and an object [1][2] [10]. The subject refers to the actor or functional user of the software; the predicates represent the methods or system transactions. As for the object, it corresponds to the classes or software components that will be implemented. In addition, the triplet represents the software requirements to be developed. *figure 1* shows the proposed triplet structure for defining quality software requirements [1][2]:

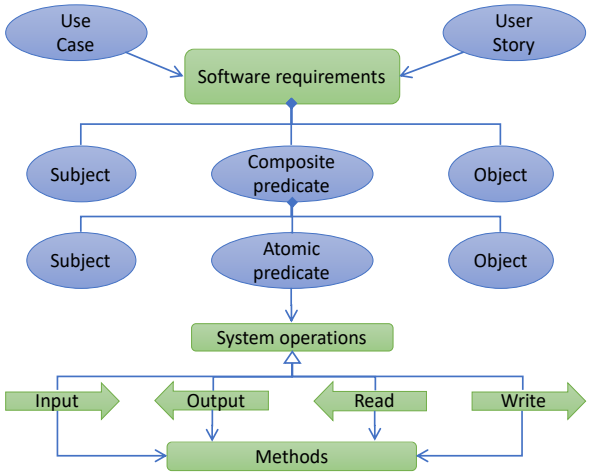

Figure 1. Triplet structure for defining quality requirements

A. Advantages of triplet structure

In the software development, there is often a lack of traceability between the different phases of the software life cycle [16]. The software specifications in analysis are not implemented in the final software. In the acceptance testing phase, it has often been found that the software does not ultimately correspond to the software specification documents. Ambler [23] argues that existing software

requirement writing techniques increase the risk of failure of software development projects. Given that the description of software requirements contains superfluous, inconsistent and ambiguous details, a large number of requirements are not implemented [15] [16]. In this case, the triplet structure could serve as a methodology for analysts, designers and developers in software requirements development for several fundamental reasons. First, it could resolve and facilitate traceability links between requirements and software development artifacts, especially in the analysis, design, coding and testing phase [10]. This is because the triplet structure provides an atomic and succinct description of software requirements expressing the unique user need without superfluous details [10]. It would facilitate the design and implementation of the software requirements specified in the analysis phase. Second, the triplet structure could solve the problems of ambiguity and inconsistency in software requirements. It emphasizes on the uniqueness, clarity and brevity of the software requirements [10]. In other words, the software requirements would be described in such a way that they do not conflict with other requirements, which would allow the semantics of the requirements to be described, avoiding the development of ambiguous requirements. It would also make it easier for the various stakeholders (managers, analysts, designers, programmers, testers) to obtain the same understanding and interpretation of software requirements [10]. Third, writing requirements as a triplet structure would allow to define good quality requirements, meeting generic criteria that quality software requirements should meet. It means that the requirements will be correct, clear and unambiguous for each stakeholder and will express the unique user need. The purpose of our technique is to propose an industry-applicable approach that reduces ambiguity and inconsistency in the process of specifying and defining software requirements. We can't go completely beyond natural language to describe requirements, since natural language is practically the only common language of the stakeholders. The triplet structure (subject, predicate, object) is a requirement writing technique that complements Use Cases and User Stories [10]. In this case, we develop a tool for generating automatically triplets from software requirements written in natural language. *Table 1* describes the characteristics of the software requirements written as a triplet structure:

Table 1- software requirements written as triplet

Characteristics of quality software requirements	Definition
Unambiguous	The stakeholders obtain the same interpretation of the requirements
Coherent	The requirements are not at odds with other requirements
Atomic	The software requirements express the unique user need
Correct	The software requirements express the real user need
Traceable	The software requirements obtained in analysis are the same in design, coding and testing
Brief	The software requirements are briefly described

B. Connections between the triplet structure and UML class diagrams

In the process of software development, the transition from analysis and design to coding seems difficult [1][2]. This is because the software operations or transactions are not sufficiently explicit in the description of the requirements in natural language. Representing the requirements as a triplet structure would also make it possible to clarify these system operations, which would be, among other things, methods in the coding or implementation phase [10]. In such an approach, a software requirement presented in the form of a triplet model, would be decomposed into several atomic predicates. The latter would therefore represent the methods that will be implemented. Finally, the triplet approach could facilitate the realization of UML class diagrams [10]. A class diagram is a model used in software engineering to represent classes, interfaces of systems and their relationships. In the domain of software requirements developments, classes constitute the software components that will be implemented. The triplet structure consists of a trio of concepts where the subject corresponds to the functional user (actor); atomic predicates correspond to system methods or transactions and to the names of associations in class diagrams [1][2][10]. As for the objects, they will correspond to the software classes in the class diagrams and to the software components that will be implemented [23]. *Table 2* describes the connections between the concepts of the triplet and UML class diagrams [1][10]:

Table 2- Mapping between the triplet structure and UML

Triplet	UML
Subject	Actor
Composite predicate	Use Case /association
Atomic predicate	Use Case scenarios/methods/system transactions or operations
Object	Objects /classes / software components/interfaces

C. Tool for detecting ambiguous requirements and identifying actors, methods and objects (classes)

We developed a tool for detecting ambiguous requirements and generating triplets from functional requirements written in natural language in order to reduce the ambiguities in the document of specification software requirements. Our tool targets the structure (subject, predicate, object). In [2][10], we assumed that the writing of software requirements is done with predicates of two arguments ***f (x, y).*** The predicate is expressed by a verb, which corresponds to the system operations or methods of the object which will be triggered following an external stimulation [10]. The "**x**" variable is a subject of the action, which corresponds to the actor. It should be noted that an actor can be a human user, a software or a software component. While the "**y**" variable is the object of the action. As for the objects of the triplet, they represent the software components or classes that will be implemented. We were inspired by the natural language analyzer offered by

Stanford NLP Software Group to determine the syntactic and semantic structure of sentences, and detect ambiguous, unclear, and imprecise sentences or requirements. The tool detects also the missing words in a sentence such as subject, verb or object. The language analyzer that we were inspired contains a class called "TagWord" which semantically identifies the words of a text written in natural language such as: subjects, verbs, and objects [24][25]. In a such perspective, we supposed that a software requirement refers to an actor (subject) that triggers an action or a system operation (verb) on an object (software component). Afterwards, we applied a number of rules to generate the triplets from software requirements written in natural language. By generating the triplets, the tool identifies respectively, as shown in *table 3*, the actors of the system, the potential methods and classes (software components) that will be implemented.

D. Potential methods and classes generated by the tool

We presented in the figure 2 the list of triplets (actor, methods and objects) generated by the proposed tool from a specification software requirements document written in natural language. The tool detects problems such as ambiguities, incompleteness, and consistency in requirements to improve the software quality. Afterwards, the tool generates from requirements a list of potential classes and methods that will be implemented. The triplet structure for representing the software requirements is simple and effective.

registrar,enters,professor
C-Reg,checks,detail
C-Reg,creates,professor
registrar,displays,message
registrar,enters,professor
C-Reg,retrieves,professor
C-Reg,displays,professor
registrar,displays,message
registrar,modifies,professor
C-Reg,validates,detail
registrar,displays,message
C-Reg,asks,course
catalog,replies,no"
C-Reg,deletes,professor
registrar,displays,error
professor,enters,ID
C-Reg,retrieves,department
C-Reg,sends,course
catalog,returns,course
C-Reg,displays,department
C-Reg,displays,course

Figure 2-List of triplets

E. Evaluation and validation of results

We tested the tool with three (3) specification software requirements documents which contain eight (8) Use Cases described in natural language in English and in French. The Use Cases split in triplets by the tool were analyzed and evaluated by human judgment based on a set of predefined criteria. These evaluation criteria are based on the syntactic and semantic quality of the triplets generated by the tool, as shown in *table 2*. Our tool generates about forty-seven (47) triplets from requirements written in natural language. Among these triplets, forty-five of these sentences are unambiguous, inconsistent, and syntactically and semantically corrects. Only two of them are syntactically and semantically incorrect. The research results showed that the generation of triplets from Uses Cases or User Stories could reduce the ambiguities in the specification of software requirements. In other words, a triplet structure for writing software specifications could help to define quality software requirements. The *table 3* presents in English and in French the experiment results.

Table 3- Experiment results in English and in French

	Syntax			Semantic			
Languages	**Wrong**	**Error**	**Correct**	**Wrong**	**Ambiguous**	**Inconsistent**	**Correct**
English	0	0	21	0	0	0	21
French	2	0	24	2	0	0	24
Total	**2**	**0**	**45**	**2**	**0**	**0**	**45**

VII. CONCLUSION AND FUTURE WORK

We proposed in this article a new technique for writing quality software requirements. This technique is a triplet structure that could help to reduce the ambiguities and inconsistencies in the document of specification software requirements, and improve software requirements quality. It is validated by the development of a tool which detects ambiguous requirements and identifies the potential methods and software components for implementation. The tool generates triplets and detects the ambiguities problems from requirements written in English, and in French. But it could be adapted to all other languages adopting the general form: ***subject, predicate, object*** **(SPO)**, such as Spanish, Haitian Creole and Italian. The limitation of our tool is that it cannot generate triplets, and point out the ambiguities problems for languages that adopt a structure ***subject, object, predicate*** **(SOP),** and ***predicate, subject, object*** (PSO). We will focus on the possibility of developing other modules that could allow the tool to exploit all the dominant combinations of language structures such as SOP and PSO. The software requirements can be functional and non-functional requirements. Another limitation of our tool is that it could not point out the ambiguities and consistencies problems for the non-functional requirements. Our future work will consist in integrating a new module which ensures that the tool could detect and reduce the ambiguities for the non-functional requirements in the document of specification software.

REFERENCES

[1] Gérançon, B., S. Trudel, R. Nkambou and S. Robert. (2021). Software Functional Sizing Automation from Requirements Written as Triplets, International Conference on Software Engineering Advances, Barcelona, Spain, 16, 23–29, 2021.

[2] Gérançon, B., S. Trudel, R. Nkambou and S. Robert. (2022). A Cognitive Model for Supporting Software Functional Sizing Automation from Requirements Written as Triplets. The 20th International Conference on Software Engineering Research & Practice, Las Vegas, USA.

[3] Boehm, B and al. (2000). Software development cost estimation approaches, A survey. Annals of software engineering, 10(1), pp.177-205.

[4] Laporte, C. Y. et April, A. (2018). *Software quality assurance*. John Wiley & Sons.

[5] April, A; Laporte.C.Y. (2015). L'assurance qualité logicielle 1, Paris : Lavoisier.

[6] K. Wiegers., J. Beatty. (2013). Software Requirements.: 3 (3erd edition) ", MICROSOFT PRESS – ISBN: 0735679665.

[7] Sadoun, D. (2014). Des spécifications en langage naturel aux spécifications formelles via une ontologie comme modèle pivot. Intelligence artificielle [cs.AI]. Université Paris Sud.

[8] Peres, F, and al. (2012). A formal framework for the formalization requirements. International of Soft Computing and Software Engineering.

[9] A. Ali, Y. Hafeez, S. Hussain and S. Yang. (2020). Role of Requirement Prioritization Technique to Improve the Quality of Highly-Configurable Systems," in *IEEE Access*, vol. 8, pp. 27549-27573, 2020, doi: 10.1109/ACCESS.2020.297138

[10] Gérançon, B. (2022). Conception et mise en œuvre d'une technique de formalisation par des triplets pour l'automatisation de la mesure de la taille fonctionnelle à partir d'exigences écrites en langage naturel. Université du Québec à Montréal (thèse de doctorat).

[11] Fraser, M.D., and al. (1991). Informal and formal requirements specification languages: bridging the gap," in IEEE Transactions on Software Engineering, vol. 17, no. 5, pp. 454-466, May 1991, doi: 10.1109/32.90448.

[12] Jacobson, I. (2003). The unified Software Development Process, Ivar Jacobson. The Journal of Object Technology, 2(4), p.7.

[13] Beck, K., D. West (2004). "User Stories in Agile Development", In Scenarios, Stories, Use Cases: Through the Systems Developments Life-Cycle.

[14] Cockburn, A. (2009). Rédiger des cas d'utilisation efficaces.

[15] Ambler, S.W. (2003). Examining the Big Requirements Up Front Approach, in line:

[16] Nuseibeh, B; and S. Easterbrook. (2000). Requirements Engineering: A Roadmap. In Proceedings of the Conference on The Future of Software Engineering, New York (USA), pages 35-46.

[17] Black, S; D. Wigg, and al. (1999). X-Ray: A Multi-Language, Industrial Strength Tool, in IWSM'99, Lac Supérieur, Canada, 39 p.

[18] Happel, H.J., and S. Seedorf (2006). Applications of ontologies in software engineering. Paper presented at International Workshop on Semantic Web Enabled Software Engineering, Athens, p. 19.

[19] Koloski, C. (1995). Méthodes et modèles de conception et d'évaluation des interfaces homme-machine

[20] Baddeley, A.D; and Hitch.G.J. (2000). Development of "IEEE Standard Glossary of Software Engineering Terminology," in IEEE Std 610.12-1990, vol., no., pp.1-84, 31 Dec. 1990, doi: 10.1109/IEEESTD.1990.101064.

[21] Roques, P. (2009). UML 2 par la pratique, Eyrolles.

[22] Trudel, S. (2012). "The COSMIC ISO 19761 functional size measurement method as a software requirements improvement mechanism", Ph.D. thesis, École de Technologie Supérieure (ETS).

[23] S.W. Ambler, "Examining the Big Requirements Up Front (BRUF) Approach", Ambysoft inc., [Online]. Available from: http://agilemodeling.com/essays/examiningBRUF.htm, [Retrieved: October, 2022].

[24] Manning, C. (2016, July). Understanding human language: Can NLP and deep learning help? In Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval (pp. 1-1).

[25] Manning, C and al. (2022). The Stanford Natural Language Processing Group. [Online]. Available from: https://nlp.stanford.edu/software/