International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878, Volume-8 Issue-3, September 2019

An Automated Tool for Detection and Tutoring of Sources of Ambiguities in Requirements Documents

Ayat Shaaban, Hanan Elazhary, Ehab Hassanein

Abstract*: Software Engineering (SE) is the application of essentials to deal with the analysis, design, development, testing, deployment and management - Software Development Life Cycle (SDLC) - of software systems. Requirements Engineering (RE) is responsible for the most critical task in the SDLC; which is transforming the requirements and wishes of the software users into complete, accurate and formal specifications. One of the main responsibilities of RE is the creation of a software requirements document that exactly, reliably, and totally defines the functional and non-functional properties of the system to be developed. At some point through the RE process, the requirements are written using a Natural Language (NL). On one hand, NLs are flexible, common, and popular. On the other hand, NLs are recognized widely as inherently ambiguous. Ambiguity is noticed in a requirement document when a piece of text is interpreted in distinct ways. This may lead to erroneous software that is too expensive to correct in later phases of the SDLC. Many tools have been developed in the literature to detect ambiguities in requirements documents. Best practices for writing requirements have also been proposed to help avoid ambiguities in the first place. The goal of this paper is to combine features from both approaches by developing the Ambiguity Checker Tutor (ACTutor), which not only detects ambiguities, but also aids in tutoring requirements engineers to apply best practices while writing requirements (rather than merely listing them). The paper is mainly concerned with proving the tutoring effectiveness of ACTutor through empirical evaluation.

***Keywords*: ambiguities, ambiguity resolution, requirements documents, software engineering, tutoring systems.**

I. INTRODUCTION

Software Engineering (SE) is an exhaustive study of the analysis, design, development, testing, deployment, and maintenance of software. It was originally introduced to address the problems of low-quality software projects. Problems arise, not only when software has reduced levels of quality, but also when it surpasses its timeline and/or its budget. Accordingly, the goal of SE is to guarantee that the software system is built correctly, on time, within budget, and most importantly according to the specified and intended requirements of the stakeholders [1].

SE generally begins with a user-request for a specific task or output. The user submits the proposal to a service provider organization. Detailed requirements are collected by numerous means including conducting interviews with the stakeholders and studying related existing systems. After requirements gathering, the development team investigates if the software can be made to satisfy all the requirements of the user. They then develop a roadmap of their plan. As per the requirements specifications and analysis, a software design is made. The implementation of software design starts in terms of writing program code in an appropriate programming language. Software testing is done while being coded by the developers; and thorough testing is conducted by experts at several levels of code such as unit testing, integration testing, system testing, in-house testing and user testing [2].

Requirements Engineering (RE) is a process of collecting and defining the services that should be provided by the system. It concentrates on determining requirements (elicitation and analysis), converting those requirements into some standard format (specification), and checking that the requirements define the system that the customer wants (validation). A software requirements document arranges out the description of the software that is to be developed as well as its purpose. It illustrates what the software is supposed to do besides how it is supposed to perform. It is written down before the actual software development work starts [3].

The requirements document is important for developers because it minimizes the amount of time and effort they have to spend to accomplish desired software goals. This also benefits the client because the lesser the development cost, the lesser the developers will charge the client. Additionally, if developed properly, the document confirms that there is less possibility of future redesigns as there is less chance of mistake on part of the developers since it gives them a clear idea of the functionalities of the software. It consequently reduces development cost by avoiding expensive modifications in later stages of the SE process.

Manuscript published on 30 September 2019
* Correspondence Author
Ayat Shaaban*, Information Systems Department, Cairo University, Cairo, Egypt. Email: ayatshaaban@live.com
Hanan Elazhary, College of Computer Science and Engineering, University of Jeddah, Jeddah, Saudi Arabia; Computers and Systems Department, Electronics Research Institute, Cairo, Egypt. Email: helazhary@uj.edu.sa
Ehab Hassanein, Information Systems Department, Cairo University, Cairo, Egypt. Email: e.ezat@fci-cu.edu.eg

The document also helps clear any communication problems between the clients and the developers since it forms the basis of mutual agreement between both parties. It also assists as the document to verify the testing processes [3]. Hence, effort should be exerted to ensure the software requirements document is specific, accurate, clear and self-consistent. Ambiguity in Natural Languages (NLs) is a major problem.

It is observed when a statement has more than one distinct meaning. The overwhelming majority of requirement specifications are written in NLs, although often accompanied by information in other notations, such as formulae and diagrams. Infrequently, one finds a completely formalized requirements document using very little NL, except as commentary. Practically and effectively, every imagination for a system or request for proposal is written in NL. Ambiguity in software requirements documents can lead to different expectations and insufficient or undesirably diverging implementations. In other words, since ambiguity can cause multiple understandings of the underlying document, it can lead to two or more implementers writing code to operate under different expectations, though each assures of having programmed the correct behavior. The ambiguity problem is aggravated if, for some reason, the author of the requirements document is not available for questioning at the time the document is being understood during development. It is quite common for the analysts who wrote a requirements document to move on to other projects and to be unavailable for questioning by the developers. In all cases, ambiguous requirements lead to confusion, high cost, wasted effort and rework [4].

Due to the seriousness of the ambiguity problem, numerous efforts in the literature have been concerned with tackling ambiguities. Some relied on the use of templates or boilerplates for writing the requirements in an attempt to minimize the introduction of ambiguities. Unfortunately, using such templates is an intimidating exhaustive process, which is not always practical [5, 6, 7]. In another approach, automated systems are developed to detect ambiguities [8, 9, 10, 11, 12, 13, 14, 15]. Nevertheless, it would be much better to avoid ambiguities in the first place. Some authors developed a set of best practices for this purpose [16, 17], but there is no means to train the requirement engineers to apply such practices. Accordingly, the goal of this paper is to combine features from both approaches by developing the Ambiguity Checker Tutor (ACTutor), which not only detects ambiguities in requirements documents, but also aids in tutoring requirements engineers to apply best practices while writing requirements by providing explanations of the sources of the detected ambiguities, and how to avoid them. To the best of our knowledge, this is the first paper to propose this approach; which is the main contribution of the paper.

The rest of the paper is organized as follows. Section II describes related work in the literature; ambiguity resolution research as well as example automated and intelligent tutoring systems. Section III describes ACTutor and its capabilities. Since this paper is mainly concerned with the tutoring capabilities of ACTutor, Section IV provides the results of its empirical evaluation in this respect. Finally, Section V presents our conclusion and directions for future research.

II. Related Work

In this section, we discuss related work in the literature. Specifically, we discuss some example research efforts in the literature for ambiguity resolution. We also provide example automated and intelligent tutoring systems.

A. Ambiguity Resolution Tools

As previously noted, many research efforts in the literature have been dedicated for ambiguity resolution in requirements documents. One research direction has been concerned with templates to help minimize the introduction of ambiguities. For example, Jain et al. [5] presented a set of templates for different kinds of requirements, but only partial implementation was demonstrated. Elazhary [6] presented a system that is designed to smooth translation of requirements between English and Arabic, based on the same templates. Alomari and Elazhary [7] provided full implementation of those templates using a set of finite state machines.

Since using such templates is an intimidating exhaustive process, which is not always practical; in another approach, automated systems are developed to detect ambiguities in requirements documents. For example, Elazhary proposed an automated requirements elicitation assistance system for English software requirements [8], and another for the Arabic language [9]. Bussel [10] developed a system for detecting ambiguous words in requirements documents. Kamsties et al. [11] suggested another inspection tool for detecting numerous ambiguities in such documents. Yang et al. [12] presented an approach for automatic detection of uncertainty in NL requirements. Bajwa et al. [13] considered syntactic ambiguities and presented a system to automatically resolve them. Gleich et al. [14] proposed a tool to automate ambiguity detection, and explain the sources of the detected ambiguities, though it was not used as a tutoring tool. Fabbrini et al. [15] presented a methodology for the analysis of NL requirements based on a quality model addressing a relevant part of the interpretation problems at the linguistic level.

Nevertheless, it would be much better to avoid ambiguities in the first place. Accordingly, some authors developed a set of best practices for this purpose. For example, Wiegers [16] proposed numerous practical techniques for improving requirements development and management processes while writing requirements documents. Elazhary [17] referred to main sources of ambiguities in the Arabic language in addition to best practices in writing Arabic user requirements, and suggested a set of best practices suitable for Arabic user requirements based on both practical experience and theoretical background. Nevertheless, there is a need for tools to train requirements engineers to apply such best practices. This is the main purpose of ACTutor, and the main contribution of this paper.

B. Automated and Intelligent Tutoring Systems

Numerous automated and Intelligent Tutoring System (ITS) have been developed through the years to help in tutoring students in the absence of teachers or for self-paced learning.

They have been developed for the tutoring of probability story problems [18], visual cognitive abilities [19], technical computer skills [20], Arabic word root extraction [21], Structured Query Language (SQL) [22], and LISP [23]. Generally, tutoring systems have been developed for the tutoring of students. In this paper, we propose applying tutoring systems for requirements engineers. It is worth noting that the tutoring system of visual cognitive abilities mentioned above [19] has been proposed for tutoring those capabilities to engineers and doctors, in addition to students.

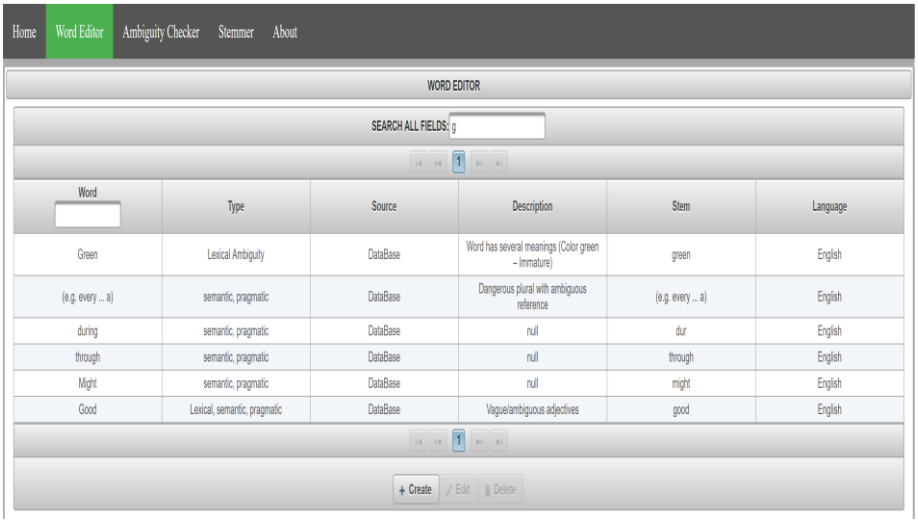

Fig. 1. The word editor.

Fig. 2. Results of processing an input document.

III. AMBIGUITY CHECKER TUTOR

This section provides a discussion of ACTutor, which was developed to be modular, extensible, and easy to use. ACTutor can process full text documents of various formats; PDF, Word, text. The input is a complete requirements text. The output is annotated requirements texts, after highlighting ambiguous words. Information regarding the sources of ambiguities are how they should be avoided are also available for requirements engineers using the tool.

A. The Word Editor Module

The word editor module depicted in Fig. 1 has three sources of words that aid in ambiguity detection:

(a) The dictionary; which contains all words that the user should use in writing the requirements document to avoid ambiguity
(b) The database, which contains all ambiguous word that the user should avoid in writing the requirements document
(c) The glossary, which contains abbreviations and domain-specific words and their definitions

The dictionary and the database aren't editable in order to keep the system consistent. Only experts can add words, to ensure they are precise. Words can be inserted, edited or deleted through the database of the system only. On the other hand, the glossary is fully editable through the user interface of the system. The user can add, edit, and/or delete words in the glossary, which displays the type, source, description, stem, and language of each word.

As shown in the figure, the user can also search based on any of those fields using the exact word or its first letter(s).

B. The Ambiguity Checker Module

The ambiguity checker module processes an input document to detect which sentences in a NL requirements document are ambiguous and, for each ambiguous sentence, identify each ambiguous word, the source of its ambiguity, and how to avoid it. There are five main types of ambiguities as discussed below [24-28].

Lexical ambiguity refers to words that inherently have several meanings such as *bound, break, call, content, continue, contract, count, direct, even, express, form, forward, function, get, job, level, name, notice, number, out, part, position, record, reference, return, set, source, special, standard, string, subject, switch, tail, throw, translate, try, under, value,* and *way*.

Referential ambiguity includes words, that may refer to numerous items such as *I, he, she, it, me, her, him, them, hers, his, its, your, their, our, herself, himself, itself, ours, ourselves, yourself, themselves, yourselves, that, theirs, these, they, this, which, who, you,* and *yours*.

Coordination ambiguity includes coordinating words that may be confusing such as *and, also, and/or, but, if and only if, if then, or,* and *unless*.

Scope ambiguity includes words with unclear scope such as *a, all, any, each, few, little, many, much, not,* and *some.*

Vagueness refers to words with unspecific meaning such as *available, common, capability, consistent, easily, easy, effective, efficient, full, general, maximum, minimum, powerful, particular, quickly, random, recent, sufficient, sufficiently, sequential, significant, simple,* and *useful*.

Fig. 2 depicts sample text with the detected ambiguities. Ambiguous words are highlighted in yellow color, while words, which are not in glossary or the dictionary, are highlighted in green. For demonstration purpose, we deactivated the dictionary while taking the screenshot. To figure out the source of ambiguity for any word and how to avoid it, the user only needs to hover over any highlighted word; and a hint message is displayed. The document can be edited (after correcting the ambiguities) and then reprocessed as needed for amendment and continuous learning.

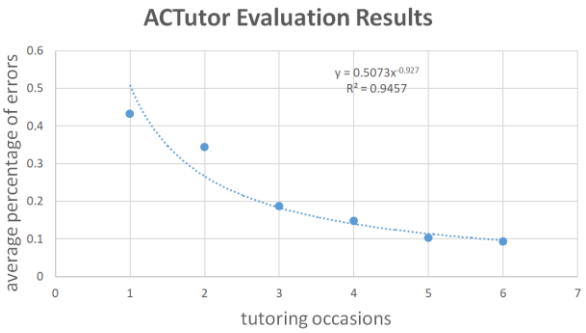

Fig. 3. Power curve demonstrating the tutoring effect of ACTutor.

IV. Empirical Evaluation

This section provides the results of the empirical evaluation of the tutoring effect of ACTutor using power curves. A power curve is a visual assessment tool that typically results due to improving the value of the assessed variable representing the learnt concept in an ITS [20].

For the purpose of the evaluation, ten volunteer novice requirements engineers participated in this study. They were given a two-hour lecture regarding how to avoid ambiguities. They were also allowed to practice using ACtutor for two days. After the two days, they were given a sample requirements document and asked to correct the ambiguities in it. The document was then processed using the tutor and they were allowed to examine the source of each ambiguity and how to avoid it. The process was repeated six times and the average percentage of errors were recorded. As shown in the power curve depicted in Fig. 3, and its corresponding log-log curve in Fig. 4, ACTutor proved to be effective in tutoring the engineers and reducing the average percentage of errors in ambiguity resolution over time.

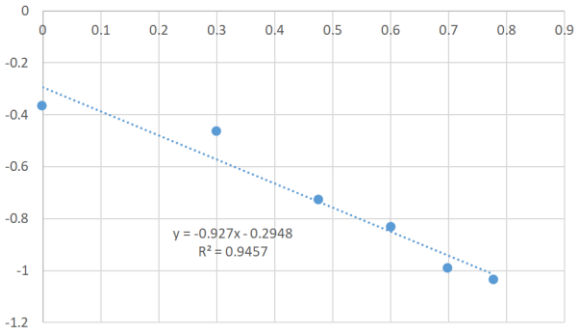

Fig. 4. A log-log curve corresponding to the power curve of Fig. 3.

V. Conclusion and Future Work

This paper proposed ACTutor, an automated tool with two goals; detecting ambiguities in requirements documents, and tutoring requirements engineers sources of ambiguities to avoid them in the future. Unlike traditional tutoring systems that have been intended mainly for students, and unlike the common practice of merely proposing best practices for avoiding ambiguities, ACTutor is intended for tutoring and training requirements engineers to apply those practices. This is the main contribution of this paper. Empirical evaluation has demonstrated the tutoring effectiveness of ACTutor. The paper considered ambiguous words that should be avoided as a proof of concept. As future work, we would consider ambiguous phrases, and longer sentence constructs for a comprehensive tutoring system.

REFERENCES

1. Sommerville I. (2015). Software Engineering. 10th ed., Pearson.
2. Software Engineering, https://economictimes.indiatimes.com/definition/software-engineering. [Online; accessed: 2018-08-01].
3. Requirement Engineering, https://medium.com/omarelgabrys-blog/requirements-engineering-introduction-part-1-6d49001526d3. [Online; accessed: 2018-08-01].
4. Tjong S. (2008). Avoiding Ambiguity in Requirements Specifications. Ph.D. thesis, University of Nottingham.

5. Jain P., Vema K., Kass A. and Vasquez R. (2009). Automated review of natural language requirements documents: Generating useful warnings with user-extensible glossaries driving a simple state machine. In 2nd India Software Engineering Conference, Pune, India.
6. Elazhary H. (2011). Translation of software requirements. International Journal of Scientific and Engineering Research, vol. 2, no. 5, pp. 1-7.
7. Alomari R. and Elazhary H. (2018). Implementation of a formal software requirements ambiguity prevention tool. International Journal of Advanced Computer Science and Applications, vol. 9, no. 8, pp. 424-432.
8. Elazhary H. (2010). REAS: An interactive semi-automated system for software requirements elicitation assistance. International Journal of Engineering Science and Technology, vol. 2, no. 5, pp. 957-961.
9. Elazhary H. (2013). An interactive system for Arabic software requirements elicitation. International Journal of Scientific & Engineering Research, vol. 4, no. 12, pp. 1941-1945.
10. Bussel D. (2009). Detecting Ambiguity in Requirements Specifications. Master's thesis, Tilburg University.
11. Kamsties E., Berry D. and Paech B. (2001). Detecting ambiguities in requirements documents using inspections. In 1st Workshop on Inspection in Software Engineering, Paris, France, pp. 68-80.
12. Yang H., De Roeck A., Gervasi V., Willis A. and Nuseibeh B. (2012). Speculative requirements: Automatic detection of uncertainty in natural language requirements. In 20th IEEE International Requirements Engineering Conference, Chicago, IL, pp. 11-20.
13. Bajwa I., Lee M. and Bordbar B. (2012). Resolving syntactic ambiguities in natural language specification of constraints. Computational linguistics and intelligent text processing. In International Conference on Intelligent Text Processing and Computational Linguistics, New Delhi, India.
14. Gleich B., Creighton O. and Kof L. (2010). Ambiguity detection: Towards a tool explaining ambiguity sources. In International Working Conference on Requirements Engineering: Foundation for Software Quality, Esen, Germany.
15. Fabbrini F., Fusani M., Gnesi S. and Lami G. (2002). The linguistic approach to the natural language requirements quality: Benefit of the use of an automatic tool. In 26th Annual NASA Goddard Software Engineering Workshop, Greenbelt, MD, USA.
16. Wiegers K. (2012). Writing high quality requirements. Requirements Management, Jama Software.
17. Elazhary H. (2016). Avoiding ambiguities in Arabic software user requirements. International Journal of Software Engineering and Its Applications, vol. 10, no. 6, pp. 141-160.
18. Khodeir N., Wanas N. and Elazhary H. (2018). Constraint-based student modeling in probability story problems with scaffolding techniques. International Journal of Emerging Technologies in Learning, vol. 13, no. 1, pp. 178-205.
19. Elazhary H. (2017). Context-aware cloud-based mobile application for assessment and training of visual cognitive abilities. International Journal of Interactive Mobile Technologies, vol. 11, no. 6, pp. 86-102.
20. Elazhary H. (2017). Cloud-based context-aware mobile intelligent tutoring system of technical computer skills. International Journal of Interactive Mobile Technologies, vol. 11, no. 4, pp. 170-185.
21. Elazhary H. and Khodeir N. (2017). A cognitive tutor of Arabic word root extraction using artificial word generation, scaffolding and self-explanation. International Journal of Emerging Technologies in Learning, vol. 12, no. 5, pp. 37-49.
22. Mitrovic A. (1998). Experiences in Implementing Constraint-Based Modeling in SQL-Tutor. In Goettl B. P., Halff H. M., Redfield C. L. and Shute V. J. (Eds.), Proceedings of the Fourth International Conference on Intelligent Tutoring Systems, San Antonio, Texas, Springer, pp.414-423.
23. Anderson J., Farrell R. and Sauers R. (1984). Learning to Program in LISP. Cognitive Science 8(2), pp. 87-130.
24. Gleich B., Creighton O., and Kof L. (2010). Ambiguity detection: Towards a tool explaining ambiguity sources. In International Working Conference on Requirements Engineering: Foundation for Software Quality, Esen, Germany.
25. Bussel D. (2009). Detecting Ambiguity in Requirements Specifications, Master's thesis, Tilburg University.
26. Berry D., Kamsties E., and Krieger M. (2003). From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity, handbook, 2003.
27. Ambiguous words, https://muse.dillfrog.com/lists/ambiguous. [Online; accessed: 2018-08-01].
28. You must not write "The system shall…", http://tynerblain.com/blog/2009/04/22/dont-use-shall/, R. L. Cauvin, [Online accessed: 2018-08-01].

AUTHORS PROFILE

Ayat Shaaban earned her B.Sc. from the Department of Computer Science and Information Systems, Akhbar Elyom Academy, 6th of October, Egypt. Currently she is a teaching assistant in the same department and is pursuing her M.Sc. degree in computers and information systems at Cairo University.

Hanan Elazhary earned her B.Sc. and M.Sc. degrees from the Department of Electronics and Communications Engineering, Cairo University. She earned her Ph.D. degree in Computer Science and Engineering from the University of Connecticut, USA. Currently, she is a professor in the College of Computer Science and Engineering, University of Jeddah, Jeddah, Saudi Arabia, and the Computers and Systems Department, Electronics Research Institute, Cairo, Egypt. She has numerous publications on requirements engineering and ambiguity resolution.

Ehab Hassanein earned a Master degree in Computer Science from Northwestern University in 1989 and a Ph. D. degree from the same university in December 1992. He is the formal Chairman of the Information Systems Department, Faculty of Computers and Information, Cairo University, Egypt. He is currently a professor in the same department.