Implementation of a Formal Software Requirements Ambiguity Prevention Tool

Rasha Alomari
Computer Science Department
Faculty of Computing & Information Technology
King Abdulaziz University, Jeddah, Saudi Arabia

Hanan Elazhary
Computer Science Department
Faculty of Computing & Information Technology
Jeddah University, Jeddah, Saudi Arabia
Computers and Systems Department
Electronics Research Institute, Cairo, Egypt

***Abstract*—The success of the software engineering process depends heavily on clear unambiguous software requirements. Ambiguity refers to the possibility to understand a requirement in more than one way. Unfortunately, ambiguity is an inherent property of the natural languages used to write the software user requirements. This could cause a final faulty system implementation, which is too expensive to correct. The basic requirements ambiguity resolution approaches in the literature are ambiguity detection, ambiguity avoidance, and ambiguity prevention. Ambiguity prevention is the least tackled approach because it requires designing formal languages and templates, which are hard to implement. The main goal of this paper is to provide full implementation of an ambiguity prevention tool and then study its effectiveness using real requirements. Towards this goal, we developed a set of Finite State Machine (FSMs) implementing templates of various requirement types. We then used Python to implement the ambiguity prevention tool based on those FSMs. We also collected a benchmark of 2460 real requirements and selected a random set of forty real requirements to test the effectiveness of the developed tool. The experiment showed that the implemented ambiguity prevention tool can prevent critical requirements ambiguity issues such as missing information or domain ambiguity. Nevertheless, there is a tradeoff between ambiguity prevention and the effort needed to write the requirements using the imposed templates.**

Keywords*—Software requirements; requirements ambiguity; natural language ambiguity; ambiguity prevention; controlled languages; finite state machines

I. Introduction

Software engineering passes through several subsequent stages. One of the preliminary stages is requirements elicitation from stakeholders. Unfortunately, elicited user requirements typically suffer from some imprecision challenging issues such as inaccuracy, inconsistency, incompleteness and ambiguity [1].

One of the most challenging issues is requirements ambiguity, which is an inherent characteristic of natural languages that are mostly used in writing software user requirements. Ambiguity occurs when an expression could have more than one way to be interpreted or understood. Consequently, it can lead to critical errors that pass through subsequent stages and end up with faulty software behavior [2, 3]. Paying attention to solving ambiguity problems in the requirements elicitation stage is much easier and less expensive than correcting later software errors. For that, many research studies in the literature attempted to tackle this problem. There is no unified terminology in the literature for classifying techniques for ambiguity resolution. Accordingly, we adopt the following definitions:

- Ambiguity avoidance: denotes using rules and best practices while writing the requirements such as those proposed by Wiegers [3, 4].
- Ambiguity prevention: refers to forcing the users to write the requirements by filling in patterns or boilerplates corresponding to different types of requirements like the work of Stalhane and Wien [5] and Arora et al. [6].
- Ambiguity detection: refers to automatically detecting ambiguities after the user requirements are written like the work of Gleich et al. [7] and Wang et al. [8].
- Ambiguity correction: refers to semi-automated tools that interact with the user to make the needed corrections such as the work of Gill et al. [9].

One of the least tackled approaches is ambiguity prevention. A major drawback is that we could hardly find a fully implemented tool for this purpose, hindering its use in practice. Additionally, there is a shortage in empirical evaluations of such techniques [10]. The reason is that this approach requires developing and implementing formal representations. Hence, this is the main concern of the paper. The rest of the paper discusses related work in the literature. After that, the ambiguity prevention tool is detailed. Next, the experiment and discussion are provided; followed by the conclusion and future work.

II. Related Work

In this section, we discuss some of the most prominent research studies in each of ambiguity avoidance, prevention, detection, and correction.

A. Ambiguity Avoidance

In ambiguity avoidance studies, the main methods used are rules and best practices. In this direction, Wiegers [3, 4] provided rules to avoid ambiguity, such as mentioning some

ambiguous words and expressions that should be avoided. Jain et al. [11] proposed a tool that can be implicitly considered an avoidance tool since it enforces requirements documentation best practices such as using standardized syntaxes and the consistent use of terminology; though it mainly falls into the ambiguity prevention class as discussed below.

B. Ambiguity Prevention

As previously noted, ambiguity prevention efforts use controlled natural languages such as templates, patterns, and boilerplates. For example, Jain et al [11] proposed a Requirements Analysis Tool (RAT) that uses templates to enforce requirements documentation best practices. RAT is comprised of a set of Finite State Machines (FSMs). It classifies the requirements into several types and then verifies that the requirements follow one of the best practice syntaxes supported by the tool. It then produces warning messages explaining where requirements are ambiguous and displays suggestions to fix them. This tool has been adopted in [12] for the Arabic language, and its full implementation is the main concern of this paper.

Denger et al. [13], on the other hand, proposed natural language patterns to be used by requirements authors when writing embedded systems requirements to prevent ambiguity. Farfeleder et al. [14] presented a tool that uses ontology-based reasoning to guide the requirements engineers and enforced this guidance by using boilerplates.

C. Ambiguity Detection

Gleich et al. [7] proposed a tool to automate the ambiguity detection process and explain the sources of detected ambiguities. It considers lexical, syntactic, semantic, and pragmatic ambiguity in addition to vagueness and language errors. This work uses part of speech tagging and regular expression search techniques for ambiguity detection. Similarly, the work of Wang et al. [15] automated the lexical ambiguity detection process focusing on overloaded and synonymous lexical ambiguity sources. The detection procedure goes through two main steps. In the first step, the C-value statistical method is used for terms extractions [16]. In the second step, the extracted terms are ranked according to the ambiguity score. The authors proposed features-based methods to estimate ambiguity scores. The ranking aims to help the requirements engineer to decide which ambiguities are more serious for time saving. Yang et al. [17] focused on one type of ambiguity, which is anaphoric ambiguity. Anaphoric ambiguity occurs when a linguistic expression may refer to two or more antecedent candidates. In this work, the authors introduced an architecture of an automated system to determine nocuous ambiguity and help requirements analysts to resolve it while discarding innocuous ambiguity that is unlikely to be misunderstood. Their approach relied on collecting human interpretations of instances of ambiguity, using heuristics to model human interpretations, and using machine learning to train the heuristics.

D. Ambiguity Correction

An example of ambiguity correction is the work of Gill et al. [9], who proposed a framework to develop semi-automatic tools for ambiguity correction in open source software requirements. They discussed some challenges in open source requirements that make it a special case.

III. Ambiguity Prevention Tool

As previously noted, ambiguity prevention approach uses controlled natural language such as templates, patterns, and boilerplates to prevent as much as possible ambiguity sources. We adopt the approach of Jain et al. [11], who uses templates, glossaries, and FSMs for this purpose. Templates are defined for six requirement classes. For each template, there is a matching FSM to analyze each requirement syntactically. Nevertheless, the authors provided merely details of the implementation of one requirements type. Hence practical use and adoption of the tool was hindered. We provide details of the implementation of all the FSMs.

In the following subsections, we explain the different requirement classes, the templates, the FSMs, the glossaries, and how the tool processes an input requirement through lexical analysis and syntactic analysis phases.

A. Requirements Classes

According to academic researchers and field experts, requirements can be classified into six classes. The six classes and their proposed syntaxes are shown and discussed below.

1) Solution requirements: This type of requirements expresses what an intended system or subsystem must do; for example:

Req01: *The system shall display completed work list items to the lab manager.*

2) Enablement requirements: Enablement requirements state what capabilities a proposed system or subsystems must provide to the users. There are two subcategories of enablement requirements. The first subcategory includes requirements that show an ability that should be provided by the software but does not decide which subsystem will provide it to the user. This is used when it is early to specify an exact ability provider; for example:

Req02: *Lab manager shall be able to create work list items.*

The second subcategory, on the other hand, includes more detailed requirements that state which system or subsystem should provide an ability to the user; for example:

Req03: *The system shall allow the lab manager to display work list items assigned to him, based on ID.*

3) Action constraint requirements: Those requirements define how the proposed system or subsystem is expected to act. There are two subcategories of action constraint requirements. The first subcategory includes requirements that state that the proposed system or some of its subsystems are allowed or not to do some action; for example:

Req04: *The loan subsystem may only delete a lender if there are no loans in the portfolio associated with this lender.*

The second subcategory, on the other hand, includes requirements that state business rules regarding how agents take some specific actions; for example:

Req05: *Only library staff may perform the loan transactions.*

4) Attribute constraint requirements: This requirements type is used to express constraints on an entity attributes or attribute values; for example:

Req06: *Search options must always be one of the followings: Price, Destination, Restaurant type, and Specific dish.*

5) Definition requirements: This category is suitable to define entities as needed; for example:

Req07: *The expected profit of a fixed rate loan is defined as the amount of interest received over the remaining life of the loan.*

6) Policy requirements: This requirements type is used to illustrate the policies that must be followed by the system; for example:

Req08: *Loan is not computed in more than one bundle.*

B. Templates and Finite State Machines

Each requirements type has a specific template in addition to a corresponding FSM to determine whether an input token stream follows the syntax. We describe the FSMs using the following variables:

- Q denotes the set of states of a given FSM based on the syntax.
- S_0 is the start state, which is the same for all FSMs.
- F is the set of final states indicating that the input token stream was based on one of the syntaxes.
- E is the set of error states indicating that the input token stream did not follow any of the syntaxes.
- S denotes the alphabet set. It includes a set of modal phrases and keywords that differentiate the various FSMs. It also includes phrases from the entity and action glossaries described below. It is the same for all FSMs.
- δ is the transition function.

1) Solution requirements FSM: The solution requirements have one accepted template as follows; its FSM is depicted in Fig. 1:

<Agent Phrase> <“shall” | “must” | “will”> <Action Phrase>

2) Enablement Requirements FSMs: Enablement requirements have two accepted templates and therefore two FSMs. The first template is as follows:

<Agent Phrase> <“shall” | “must” | “will”> <“be able to”> <Action Phrase>

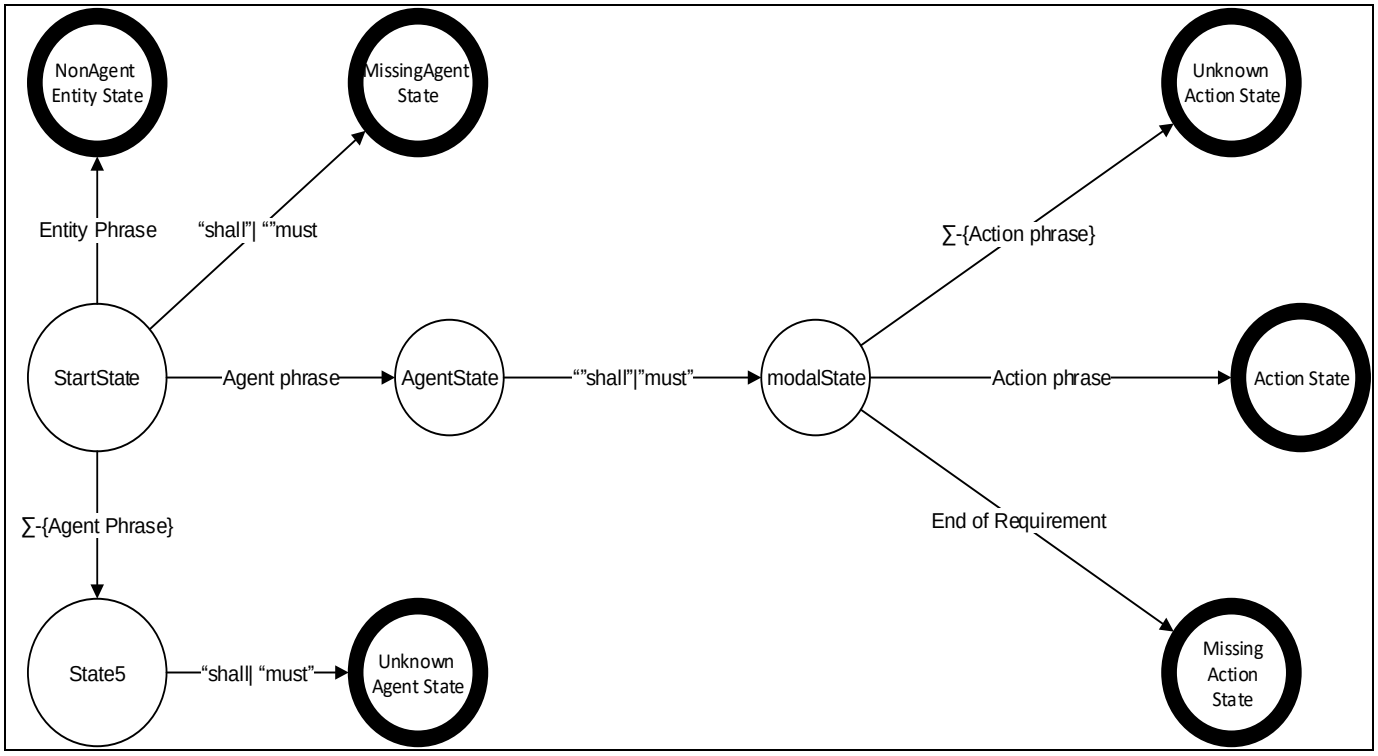

Fig. 1. Solution Requirements FSM.

The corresponding FSM is depicted in Fig. 2. It can be described as follows:

- Q = {Start State, Action State, Modal State, Agent State, Missing Agent State, Missing Action State, Unknown Action State, Unknown Agent State, Non-Agent Entity State}
- F = {Action State}
- E = {Non-Agent Entity State, Missing Agent State, Unknown Action State, Missing Action State, Unknown Agent State}

<Agent Phrase> <"shall" "must"|"will"> <"allow" | "permit"> <Agent Phrase><"to"> <Action Phrase>

The second accepted template of enablement requirements is as follows:

The corresponding FSM is depicted in Fig. 3. It can be described as follows:

- Q = {Start State, Action State, Modal State, Agent State, Missing Agent State, Missing Action State, Unknown Action State, Unknown Agent State, Non-Agent Entity State}
- E = {Non-Agent Entity State, Missing Agent State, Unknown Action State, Missing Action State, Unknown Agent State}
- F = {Action State}

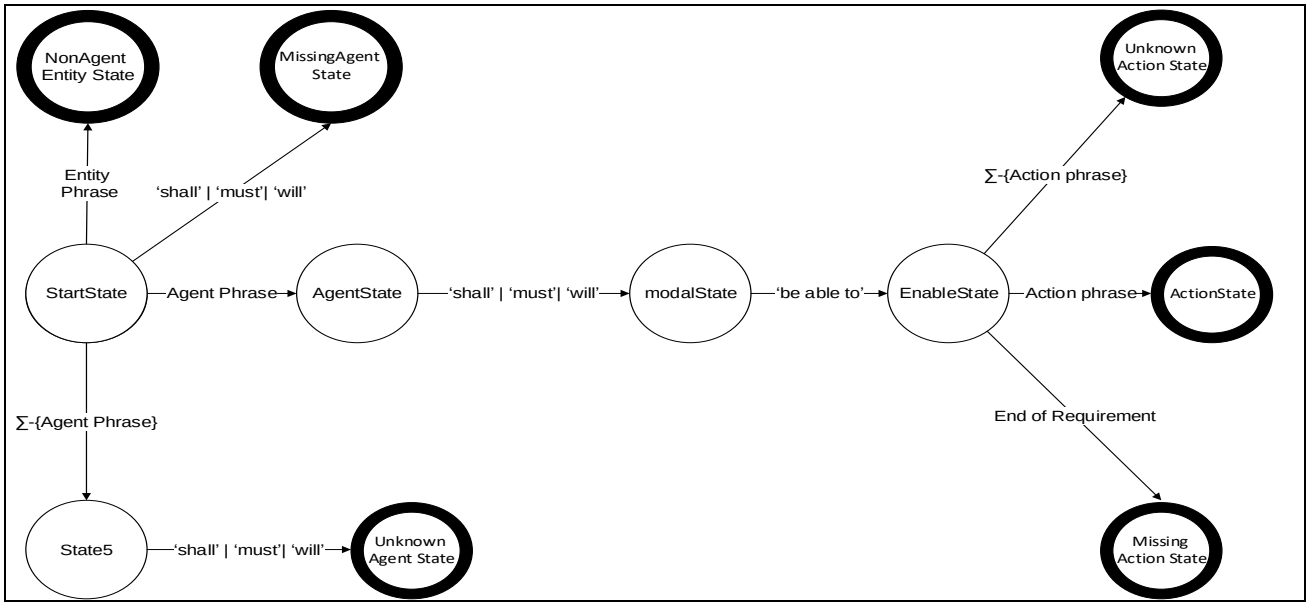

Fig. 2. Enablement Requirements FSM (1).

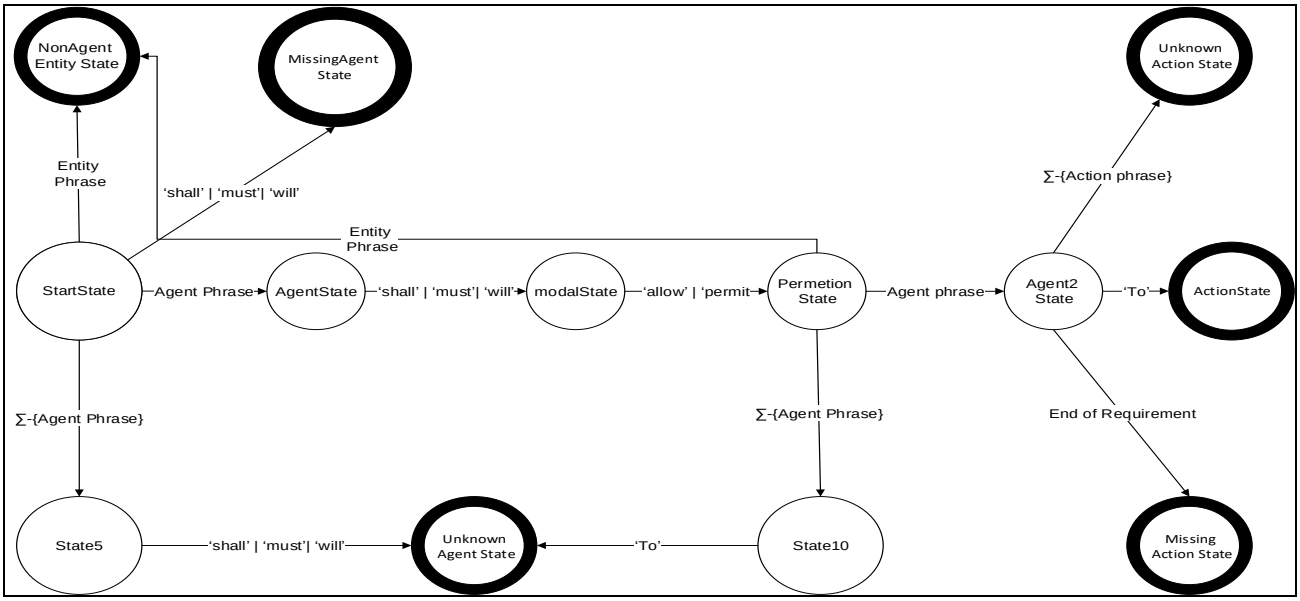

Fig. 3. Enablement Requirements FSM (2).

3) Action constraint requirements FSMs: Action constraint requirements have two accepted templates. The first one is as follows:

<Agent Phrase> <"shall" | "will" | "may"> <"only" | "not"> <Action Phrase> <"when" | "if"> <condition>

The corresponding FSM is depicted in Fig. 4. It can be described as follows:

- Q = {Start State, Action State, Modal State, Agent State, Missing Agent State, Missing Action State, Unknown Action State, Unknown Agent State, Non-Agent Entity State}
- F = {Action State}
- E = {Non-Agent Entity State, Missing Agent State, Unknown Action State, Missing Action State, Unknown Agent State}

The second accepted template of action constraint requirements is as follows; the corresponding FSM is shown in Fig. 5:

"Only" <Agent Phrase> <"may"| "may be"> <Action Phrase>

4) Attribute Constraint Requirements FSM: Attribute constraint requirements have one accepted template:

<Entity Phrase | Agent Phrase> "must" <"always"| "never" | "not"> <"be" | "have"> <Value Phrase>

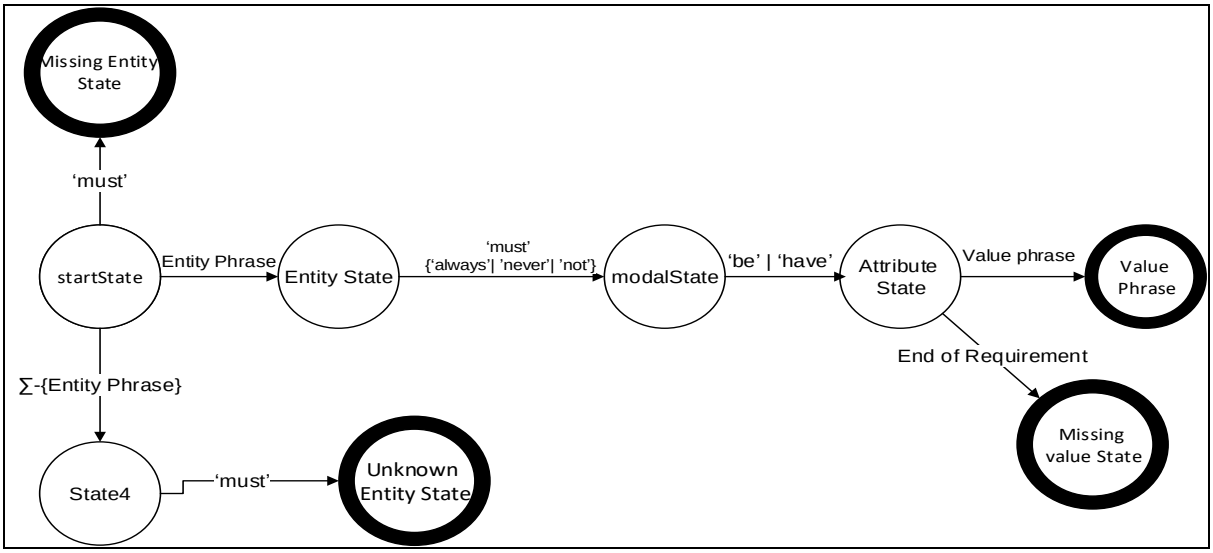

Fig. 4. Action Constraint Requirements FSM (1).

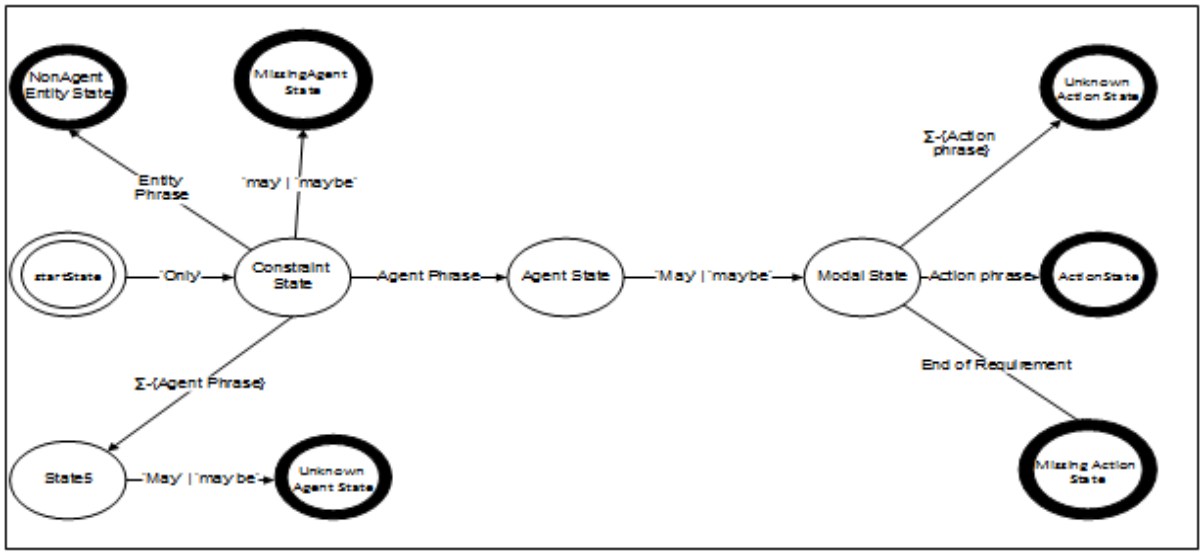

Fig. 5. Action Constraint Requirements FSM (2).

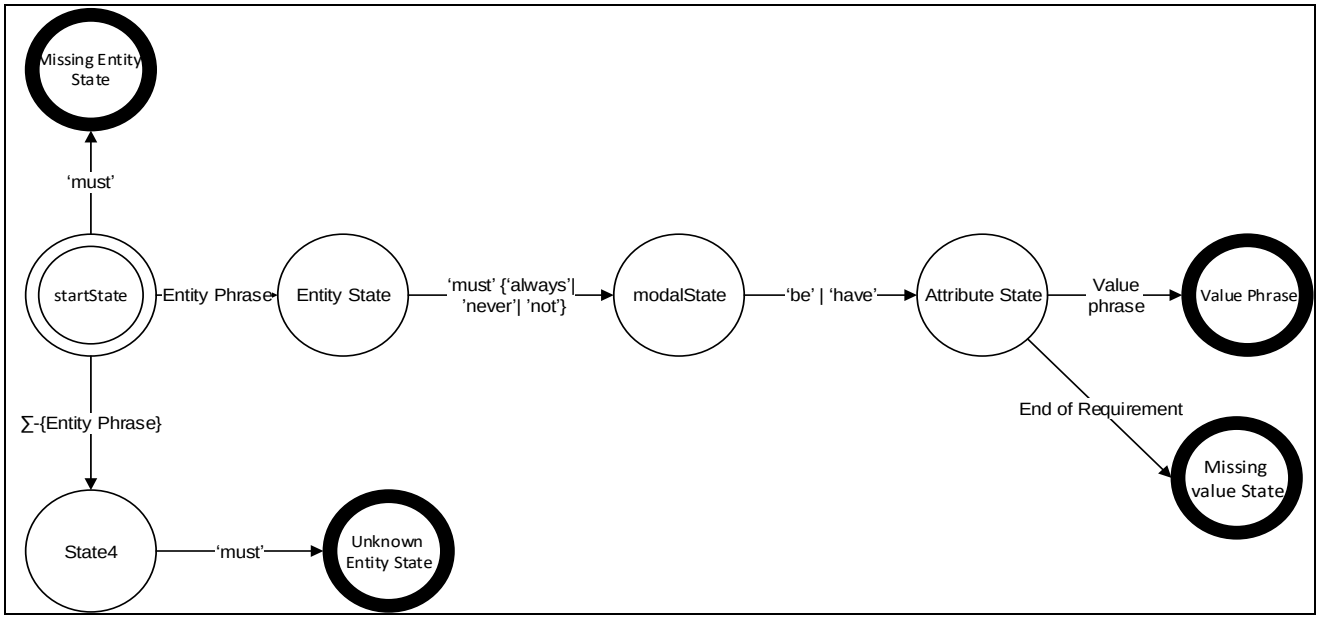

Fig. 6. Attribute Constraint Requirements FSM.

The corresponding FSM is depicted in Fig. 6. It can be described as follows:

- Q = {Start State, Action State, Modal State, Agent State, Missing Agent State, Missing Action State, Unknown Action State, Unknown Agent State, Non-Agent Entity State}
- F = {Action State}
- E = {Non-Agent Entity State, Missing Agent State, Unknown Action State, Missing Action State, Unknown Agent State}

5) Definition requirements FSM: Definition requirements have one accepted template as follows:

<Entity Phrase | Agent Phrase> <"is" | "will be"> <"defined as" | "classified as"> < Entity Phrase>

The corresponding FSM is depicted in Fig. 7. It can be described as follows:

- Q = {Start State, Action State, Modal State, Agent State, Missing Agent State, Missing Action State, Unknown Action State, Unknown Agent State, Non-Agent Entity State}
- F = {Action State}.
- E = {Non-Agent Entity State, Missing Agent State, Unknown Action State, Missing Action State, Unknown Agent State}

6) Policy requirements FSM: Policy requirements have one accepted template as follows:

<Entity Phrase | Agent Phrase> <"is | "is not"> <Action Phrase>

The corresponding FSM is depicted in Fig. 8. It can be described as follows:

- Q = {Start State, Action State, Modal State, Agent State, Missing Agent State, Missing Action State, Unknown Action State, Unknown Agent State, Non-Agent Entity State}
- F = {Action State}.
- E = {Non-Agent Entity State, Missing Agent State, Unknown Action State, Missing Action State, Unknown Agent State}.

C. The Glossaries

The glossaries are an essential component of the implemented tool. The program consults user-defined glossaries to determine whether an input requirement uses predefined accepted terminology or not. Moreover, glossaries are necessary for lexical and syntactic analysis as described below.

We use two glossaries: an entity glossary and an action glossary. The entity glossary contains an entry for each accepted entity in the requirements document. Table I shows an example of an entity glossary content. The glossary determines whether each entity is an agent or not. An agent entity is the one that can do an action such as 'booking user' or 'library stuff', while a non-agent entity is an entity that does not perform an action such as 'the loan'. An action glossary, on the other hand, contains an entry for every accepted action phrase. Table II shows an example content of an action glossary.

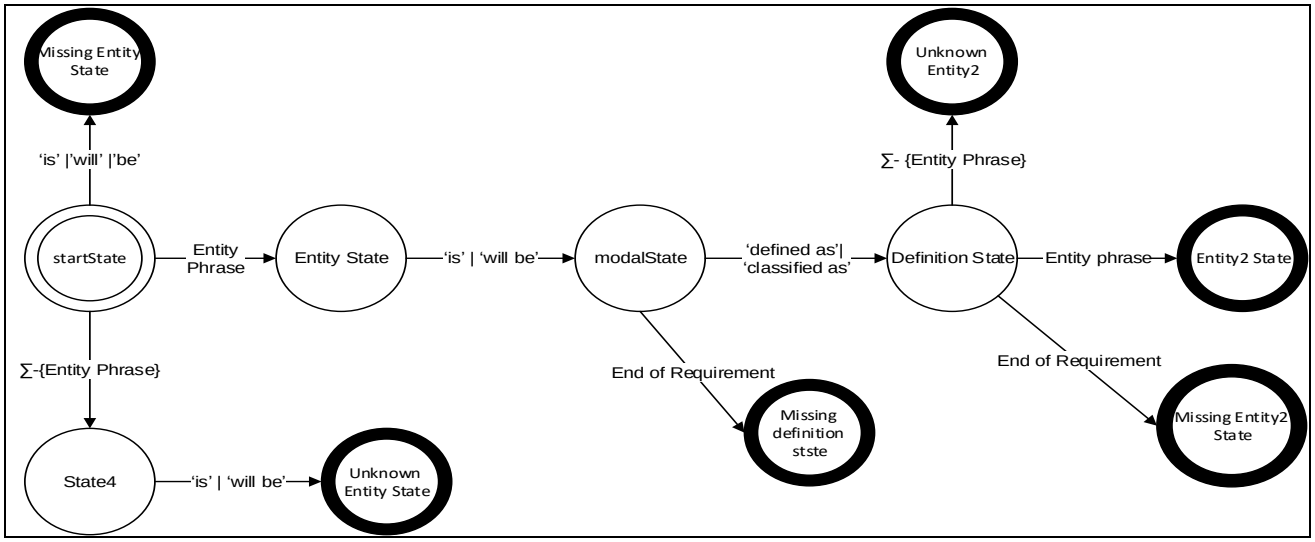

Fig. 7. Definition Requirements FSM.

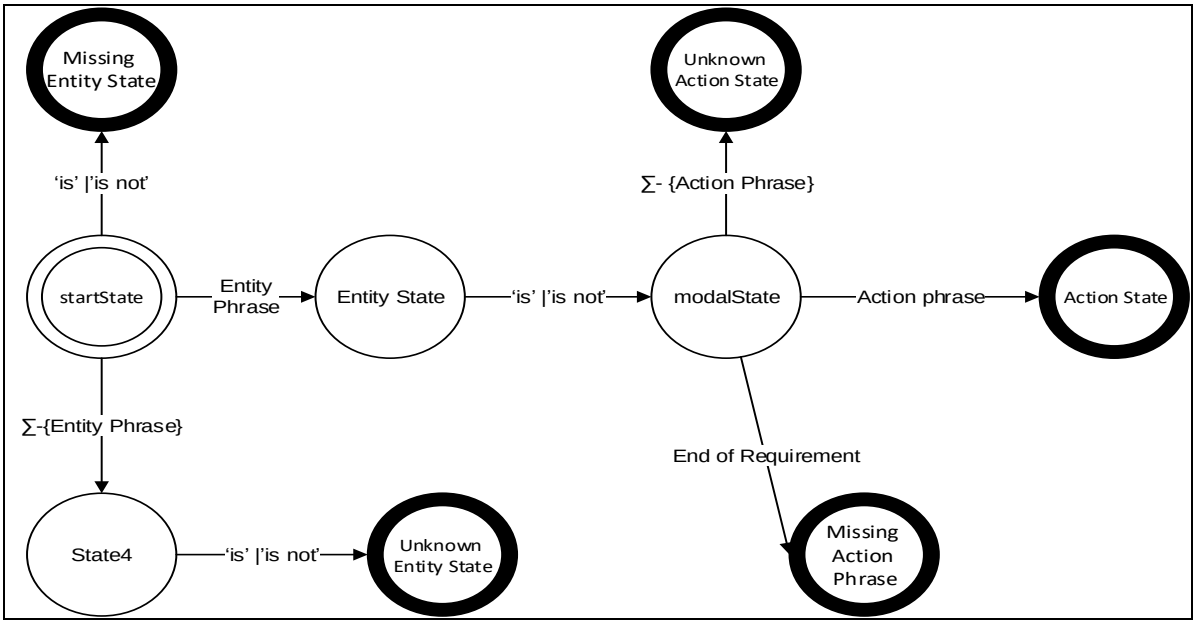

Fig. 8. Policy Requirements FSM.

TABLE I. EXAMPLE OF ENTITY GLOSSARY CONTENT

Entity Descriptor	**Explanation**	**Is Agent**
Borrower	The recipient of money from a lender. Borrowers may receive loans jointly; that is, each loan may have multiple borrowers.	Yes
HR User	User from human resource department	Yes
Protocol	the exact methodology used to analyze samples	No
ProdID	Product Identification; unique identifier of each product	No
Product Sample	A small amount of product taken from a specific product	No

TABLE II. EXAMPLE OF ACTION GLOSSARY CONTENT

Action Descriptor	**Explanation**
process orders	Action for processing orders
Display	Rendering an item on screen
send contracts data	Action for transfer of contract data
inform administrator	Action for sending e-mail notification to administrator
process payroll	Action for processing of payroll

TABLE III. TOKEN TYPES AND TAGS

Token Type	**Tag**
Label	Lbl
Entity phrase	En
Agent phrase	Ag
Action phrase	Ac
Modal phrase	Mod
Constant phrase	Const
Unknown	Un

D. Lexical Analysis

In the lexical analysis phase, the program consults the glossaries to tokenize a given requirement statement and then classify and tag each token into "entity phrase", "agent phrase", "action phrase", "modal phrase", "constant phrase", or "unknowns" as depicted in Table III. The term constant phrase indicates phrases that do not fall into any of the other token types such as "be able to", "only", and "permit". Stop words such as "the", "a", "an", "for", "too" and "up" are ignored in the process.

As an example, to clarify the tokenization, classification, and tagging processes, consider the following requirement statement:

Req00: *The user must be able to display the PDF rendition of associated documents.*

The output of lexical analysis will be as follows:

Req00	User	must	be able to	display the PDF rendition of associated documents
Lbl	Ag	Mod	Const	Ac

E. Syntactic Analysis

The tokenized tagged requirement from the previous phase is input to the syntactic analysis phase. Syntactic analysis passes through the following process:

1) Reading each tokenized requirement.

2) Classifying the requirement into one of the six requirement classes depending on the modal phrase. The goal of this step is to decide which of the FSMs to use.

3) Using the suitable FSM to check whether the requirement statement follows the corresponding accepted syntax and to generate useful warning massage as needed.

According to the final state the parser reaches, the user receives useful warnings as needed. Fig. 9 shows a screenshot of the tool depicting example input and output.

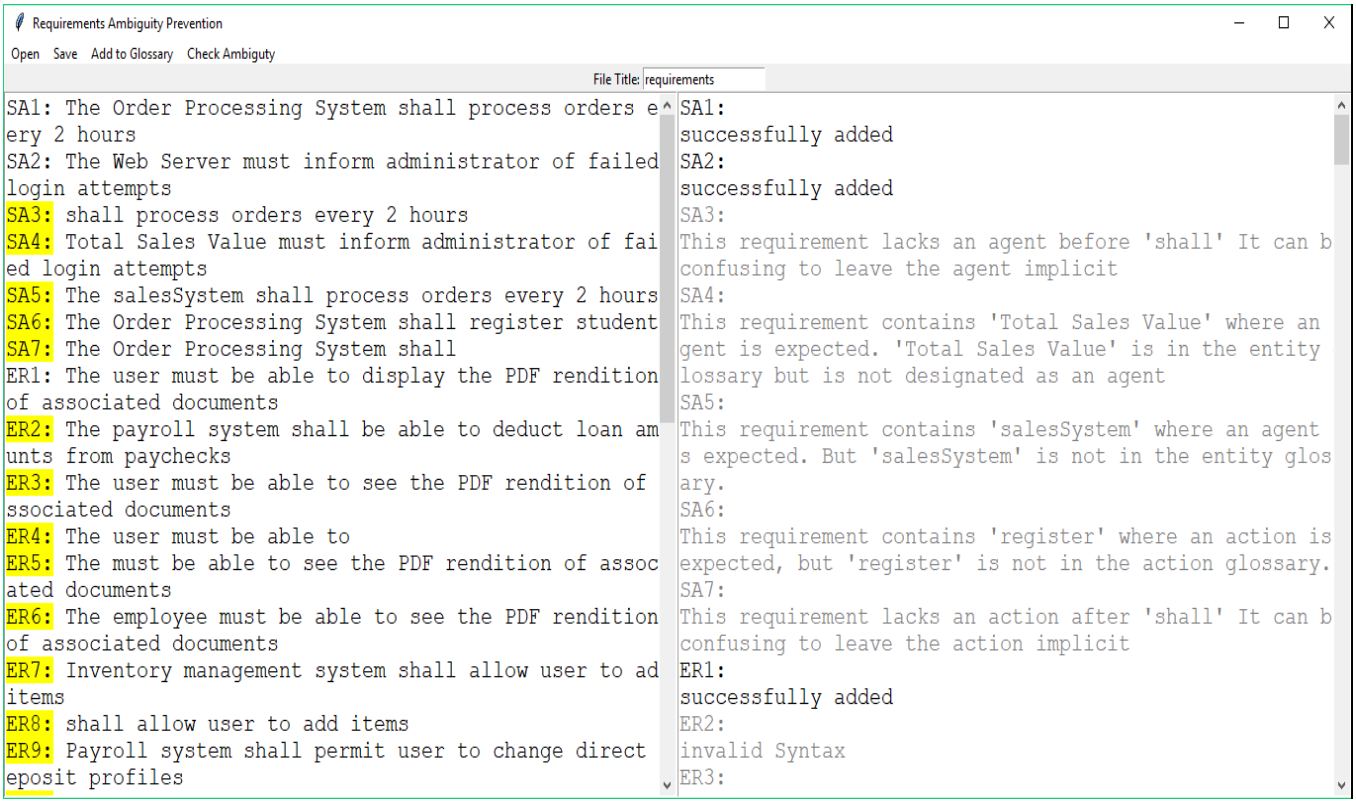

Fig. 9. A Screenshot of the Ambiguity Avoidance Tool; Example Input and Output.

IV. Experiment and Discussion

A. Experimental Settings

We used Python version 3.6 to implement the ambiguity prevention tool. Then, we built a benchmark of 2460 real requirements. From the benchmark, we selected a random sample of forty real requirements. We classified each requirement in the sample into one of the six requirement classes mentioned above. We then transformed each classified requirement into the corresponding template and defined entities, agents, and actions in the glossaries. The purpose of this process is to emulate a real user writing the requirements before processing them through the tool.

B. Results and Discussion

From the experiment, it was clear that this approach can prevent some types of requirements ambiguity. Example issues that could be prevented using this approach are: missing information like missing an agent or missing an action; domain ambiguity like an unknown agent or an unknown entity; and non-best practices syntax like missing an action or an invalid syntax.

But on the other side, it was clear that classification and transformation processes are not straightforward. For example, some requirements had to be split into two requirements of different classes and templates.

Moreover, it was clear that the overall requirements writing process consumes more time and effort than using an un-controlled natural language. In other words, there is a tradeoff between the effort needed to write the requirements following the predefined templates and ambiguity avoidance.

V. Conclusion and Future Work

This paper presented details of a full implementation of a software requirements ambiguity prevention tool. This tool classifies the software requirements into one of six classes: solution, enablement, action constraint, attribute constraint, definition, or policy requirements. For each requirement class, there is an accepted defined template. To check whether the requirements adhere to the correct templates, the tool uses a FSM for each template.

We used Python to implement and test this approach. We selected forty random requirements sample out of 2460 real software requirements. We noted that the selected approach has some advantages and disadvantages as discussed above. But to judge this approach precisely, we need to compare it with other prominent approaches in our future work. It is important because we need to compare different approaches from some aspects such as: effectiveness in term of types and number of ambiguities resolved. We also need to compare the usability of the different approaches.

References

[1] G. Sandhu, "Analysis of modeling techniques used for translating natural language specification into formal software requirements," International Journal of Computer Applications, vol. 113, no. 1, 2015.

[2] H. Elazhary, "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, 2010.

[3] K. Wiegers, "Karl Wiegers describes 10 requirements traps to avoid," Software Testing & Quality Engineering, vol. 2, no. 1, 2000.

[4] K. Wiegers, "Writing quality requirements," Software Development, vol. 7, no. 5, pp. 44-48, 1999.

[5] T. Stalhane and T. Wien, "The DODT tool applied to sub-sea software," in 2014 IEEE 22nd International Requirements Engineering Conference, 2014, pp. 420-427.

[6] C. Arora, M. Sabetzadeh, L. Briand, F. Zimmer, and R. Gnaga, "Automatic checking of conformance to requirement boilerplates via text chunking: An industrial case study," in 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 35-44.

[7] B. Gleich, O. Creighton, and L. Kof, "Ambiguity detection: Towards a tool explaining ambiguity sources," Requirements Engineering: Foundation for Software Quality, pp. 218-232, 2010.

[8] Y. Wang, I. L. M. Gutiérrez, K. Winbladh, and H. Fang, "Automatic detection of ambiguous terminology for software requirements," in Natural Language Processing and Information Systems: Springer, 2013, pp. 25-37.

[9] K. D. Gill, A. Raza, A. M. Zaidi, and M. M. Kiani, "Semi-automation for ambiguity resolution in open source software requirements," in 2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering, 2014, pp. 1-6.

[10] M. Bano, "Addressing the challenges of requirements ambiguity: A review of empirical literature," in 2015 IEEE 5th International Workshop on Empirical Requirements Engineering (EmpiRE), 2015, pp. 21-24.

[11] P. Jain, K. Verma, A. Kass, and R. G. Vasquez, "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, 2009, pp. 37-46.

[12] H. Elazhary, "Translation of Software Requirements," International Journal of Scientific and Engineering Research, vol. 2, no. 5, pp. 1-7, 2011.

[13] C. Denger, D. M. Berry, and E. Kamsties, "Higher quality requirements specifications through natural language patterns," in IEEE International Conference on Software: Science, Technology and Engineering, 2003, pp. 80-90.

[14] S. Farfeleder, T. Moser, A. Krall, T. Stålhane, I. Omoronyia, and H. Zojer, "Ontology-driven guidance for requirements elicitation," The semanic web: Research and applications, pp. 212-226, 2011.

[15] Y. Wang, I. L. M. Gutiérrez, K. Winbladh, and H. Fang, "Automatic detection of ambiguous terminology for software requirements," in International Conference on Application of Natural Language to Information Systems, 2013, pp. 25-37.

[16] K. T. Frantzi and S. Ananiadou, "Extracting nested collocations," in 16th Conference on Computational Linguistics, vol. 1, 1996, pp. 41-46.

[17] H. Yang, A. De Roeck, V. Gervasi, A. Willis, and B. Nuseibeh, "Analysing anaphoric ambiguity in natural language requirements," Requirements engineering, vol. 16, no. 3, p. 163, 2011.