Automatic Detection of Ambiguous Terminology for Software Requirements

Yue Wang, Irene L. Manotas Gutiérrez, Kristina Winbladh, and Hui Fang

Department of Electrical and Computer Engineering,
University of Delaware,
Newark, DE 19716
{wangyue,imanotas,winbladh,hfang}@udel.edu

Abstract. Identifying ambiguous requirements is an important aspect of software development, as it prevents design and implementation errors that are costly to correct. Unfortunately, few efforts have been made to automatically solve the problem. In this paper, we study the problem of lexical ambiguity detection and propose methods that can automatically identify potentially ambiguous concepts in software requirement specifications. Specifically, we focus on two types of lexical ambiguities, i.e., *Overloaded* and *Synonymous* ambiguity. Experiment results over four real-world software requirement collections show that the proposed methods are effective in detecting ambiguous terminology.

Keywords: Ambiguity detection, Software requirements, Overloaded ambiguity, Synonymous ambiguity

1 Introduction

A Software Requirements Specification (SRS) describes the required behaviour of a software product, and is often specified as a set of necessary requirements for project development. An ideal SRS should clearly state the requirements without introducing any ambiguities. Unfortunately, it is impossible to avoid the ambiguous SRSs since they are often described using natural languages.

A requirement is ambiguous if it can be interpreted in multiple ways. Ambiguous requirements can be a major problem in software development [4]. Project participants tend to subconsciously disambiguate requirements based on their own understanding without realizing that they are ambiguous. As a result, different interpretations often remain undiscovered until later stages of the software life-cycle, when design and implementation choices materialize the specific interpretations. It costs 50-200 times as much to correct an error late in a software project compared to when it was introduced [3].

One possible way of preventing ambiguous requirements is through manual inspection [17], which clearly is time-consuming and error prone. Consequently, it is important to study how to automatically detect ambiguous requirements in software requirement specifications (SRS).

Establishing a consistent usage of terminology early on in a project is imperative as it provides a vocabulary for the project and can greatly reduce misunderstandings. In this paper, we focus on the problem of lexical ambiguity detection. Specifically, we aim to detect terminology misuse such as overloaded and synonymous concepts. We use the word *concept* instead of *term*, because we consider both terms and phrases. A concept is *overloaded* if it refers to different semantic meanings and it is *synonymous* if several different concepts are used interchangeably to refer to the same semantic meaning (see Fig. 1). Note that overloaded concepts include both homonyms and polysemy.

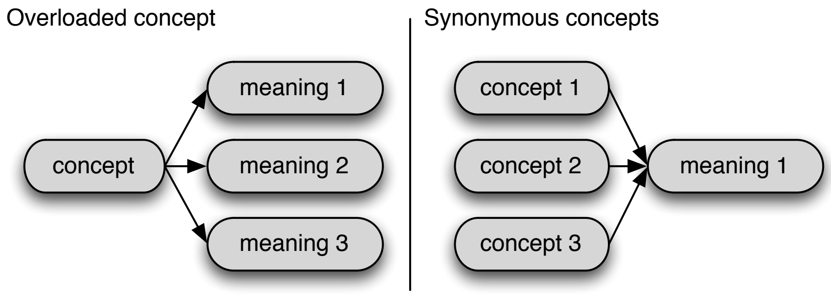

Fig. 1. Overloaded and synonymous concepts.

We propose to formulate the problem as a ranking problem that ranks all the important concepts from a SRS based on their ambiguity scores. The ranked list of concepts is expected to help requirement engineers to more efficiently identify ambiguous concepts and revise the SRS accordingly. One advantage of formulating the problem this way is to allow requirements engineers to decide how many concepts they want to go through based on their own situations. For example, some engineers may want to catch all ambiguous concepts while others may only have limited time to correct the most ambiguous ones. Once the ambiguous concepts are identified and rephrased, the SRS would have higher quality and can be better used in the subsequent stages of the project.

Specifically, we propose two feature-based methods that can rank the concepts based on their overloaded and synonymous ambiguities respectively. Experiments are conducted over four data sets with real-world SRSs. These data sets cover different types and scales of software systems. Results show that the proposed methods are effective in detecting both overloaded concepts and synonyms.

2 Related Work

Requirements ambiguities can be avoided by using formal languages to specify the requirements. Formal languages use mathematical notations and syntax to specify requirements precisely and can be used to check the requirements for inconsistencies and other problems. A non-extensive list of formal approaches include approaches that use logic-based, state-based, event-based, and algebraic-based representations [5, 6, 8, 22]. Although formal specification languages do avoid ambiguities, there are some limitations in using them. One limitation is that formal notations require more efforts from requirements engineers and other participants in creating and reviewing requirements. Another limitation is that although a formally specified requirement might be free of ambiguities, it could still be incorrect as it has been translated from an informal requirement at some point. That is, the same disambiguating assumptions can be made when translating informal requirements into the formal notation as when leaving the requirements in their informal representation and using them in subsequent development activities. It is therefore important to disambiguate the language used in the informal representation prior to using a formal notation.

A common approach to handle ambiguous requirements problem in SRSs is the use of a project glossary. The creation of a project glossary generally occurs during domain understanding and requirements elicitation. Although a project glossary can play an essential role in a software project, there is usually no quality checks on the glossary. It turns out that many glossaries are rather weak in the sense that they do not cover the terminology that is actually used in a specification and the synonymous and overloaded concepts are not recognized and marked [25]. Chantree et al. present an interesting approach with a focus on identifying ambiguities that are likely to lead to misunderstandings [4]. Others have worked on resolving requirements ambiguities that are likely problematic to requirements engineers [2, 18]. Our work differs in that we focus on terminology consistency and specifically on reducing the ambiguity that can result from terminology misuse.

Some studies tried to combine machine learning and NLP techniques to identify ambiguous requirements at the sentence level [10, 15] . On the contrary, this paper focused on detecting the ambiguous requirements from at the concept level.

Our work is closely related to the word sense disambiguation problem, which determines the appropriate sense of a word given its context and the senses often are defined in a dictionary. However, our work focuses on a different problem, and aims to detect whether a concept is ambiguous in a requirements document. The problem is more challenging than in the general domain since the definition of the ambiguity is more subtle. First, the problem is domain-specific, and there is no dictionary available for each domain to describe possible senses of every concept, which requires us to automatically identify possible senses by ourselves. Second, the definition of ambiguity in SRSs is not well defined, and relies highly on the context of the concepts. A concept may be used ambiguously in the requirements of one project, but not in other projects.

3 Problem Formulation

A SRS is ambiguous if it can be interpreted in more than one ways [2]. There are many different types of ambiguities, and here we focus on lexical ambiguities. Lexical ambiguities can be classified into *overloaded ambiguity* and *synonymous ambiguity* [25], as shown in Figure 1. We define an overloaded ambiguity to be a concept that has lost its specificity in the particular document. For example, consider the concepts *user*, *guest user*, and *verified user* in a SRS. In cases where only *user* is used in the SRS, a reader may not be able to distinguish which kind of user is intended. In contrast to overloaded ambiguity, synonymous ambiguity is when multiple concepts refer to the same semantic meaning. For instance, in the SRS of a testing gateway system, the concepts *system* and *testing gateway* both refer to the system to be developed. As a result, requirement engineers could use both concepts in the SRS without realizing the potential for conflicts and misunderstandings.

To detect ambiguous concepts from a SRS collection, we first use C-value method [24, 7] to extract candidate concepts, and then propose to rank the extracted concepts or concept pairs based on their degree of ambiguity. In particular, for overloaded ambiguity detection, concepts should be ranked based on the likelihood that a concept has multiple interpretations, while for synonymous ambiguity detection, concept pairs are ranked based on the likelihood that they represent the same meaning. The ranked lists are expected to help requirements engineers focus on the concepts that are most likely to be ambiguous so that they can quickly identify the places that need clarification.

The key challenge here is how to estimate the ambiguity score for a concept or a concept pair. We focus on identifying useful features that could be used to identify each type of ambiguities. For overloaded ambiguities, the features are mostly related to the *context* of a concept, i.e., words that occur before and after the concept in the same sentence. For synonymous ambiguity detection, the features are based on not only context but also patterns and content of the candidate pairs. With the identified features, we then propose a possible solution to combine them and learn the ambiguity scores for the concepts or concept pairs. Details are provided in the following sections.

4 Overloaded Ambiguity Based Ranking

As defined previously, overloaded ambiguities lead to a "one-to-many" mapping from concepts to semantic meanings. Since the context of a concept is closely related to its semantic meaning, the degree of ambiguity of a concept should be determined by how diverse its context is. We study the following features that measure the diversity of the context for a concept.

– **Concept frequency:** Given a concept, this feature computes the frequency of the concept in all the SRSs. The intuition is that a concept is more likely to cause an overload ambiguity when it occurs more frequently in the collection.
– **Context diversity:** For a given concept, the feature measures how diversified its contexts are. We define a context of a concept as a set of words that occur in the same sentence as the concept. If the concept is overloaded, its context should cover different meanings for the sub-layer entities. Therefore, the diversity score should be high. On the other hand, the entity that the concept refers to should be consistent among different contexts, which means the context diversity should be low. The context diversity score of a concept is computed as the inverse of the average cosine similarity among all its contexts.
– **Number of clusters in the context:** Clustering is one possible way of partitioning contexts of a concept into different groups with similar meanings. Thus, the number of clusters could be a good indicator of the degree of ambiguity of the concept. In this paper, we use hierarchical agglomerative clustering method [12]. There are multiple ways for clustering. During the training stage of our experiment, we tried single-link, complete-link and centroid HAC algorithm. The results suggested that the single link algorithm consistently outperform than the other algorithms. Therefore, it is chosen as the final method. We keep grouping similar contexts together until it reaches the stopping criterion, i.e., when the minimum similarity between each group is smaller than a *similarity boundary*.
– **Inter-cluster distance:** It measures the average distance among different clusters. The intuition is that when a concept is ambiguous, its context clusters would cover different information, which leads to higher inter-cluster distance. The distance is

computed as the inverse of the similarity, which can be computed using cosine similarity based on the context.

We now discuss how to combine all the features. Since each feature can be used individually to rank concepts, we can then compute the ambiguity score of a concept based on its ranking positions using each of the features. The concepts are then ranked based on these scores.

Formally, c denotes a concept, $AS_O(c)$ denotes the overloaded ambiguity score of the concept, and $f_i(c)$ is the value of feature f_i for concept c. We can then have:

$$AS_O(c) = \sum \alpha_i \cdot PS(f_i(c))$$

where α_i is the weight of the result of each feature f_i and $\sum \alpha_i = 1$. The weights can be learned from a training set. $PS(x)$ is the relative position score of each feature and can be computed as:

$$PS(f_i(c)) = 1 - \frac{PositionInFeature(c, f_i) - 1}{\#TotalConcepts} \quad (1)$$

where the PositionInFeature is the ranking of concept c in feature f.

Note that there could be other ways of combining these features. We choose to use the relative value instead of the absolute score from each feature is because we want to make the results from different features more comparable.

5 Synonymous Ambiguity Based Ranking

A synonymous ambiguity is caused by a "many-to-one" mapping between concepts and semantic meanings. We identify the following features that can be used to identify synonymous ambiguity:

- **Context-based similarity:** It computes the average similarity of contexts for each pair of concepts. However, it is possible that two concepts have similar contexts but are not synonymous. For example, concepts *user ID* and *password* may co-occur frequently in a SRS collection, but they are not considered as synonymous. To solve this problem, we propose to consider only concept pairs that do not occur in the same sentence when computing the context similarities. Thus, the context-based similarity of two concepts c_i and c_j can be computed as follows:

  $$Sim_C(c_i, c_j) = \frac{\sum_{x \in UC(c_i|c_j), y \in UC(c_j|c_i)} Similarity(x, y)}{|UC(c_i|c_j)| \times |UC(c_j|c_i)|}$$

  where $UC(c_i|c_j)$ is a set of context for concept c_i that do not contain concept c_j. $Similarity(x, y)$ measures the similarity between two contexts and is computed using cosine similarity.
- **Pattern-based similarity:** Pattern-based features have been used to detect the semantic relationship in large text corpora [9, 19, 14]. We follow a similar strategy to detect synonym pairs in this paper. In particular, we start with a set of known pairs of synonymous concepts, and then retrieve the sentences that mention both concepts. We then identify patterns, i.e., common phrases or terms, and these patterns will then use to retrieve more candidate pairs. The process is repeated until no more new patterns can be found.

If concept c_i and c_j follow the discovered pattern P, then we have

$$Sim_P(c_i, c_j) = Sim_P(c_j, c_i) = 1.$$

Following the proposed methods, we are able to find the following patterns:
- c_1 **abbreviated** c_2
- c_1 (c_2)
- c_1**, also known as** c_2
- c_1**, a.k.a.** c_2

– **Textual-based Similarity:** A synonymous concept pair reflects the same semantic meaning, so it is likely that their textual similarity is higher than other pairs. For example, concepts *account reference number* and *original account number* both refer to the number assigned to a user when opening an account. Thus, we have

$$Sim_T(c_i, c_j) = CosineSimilarity(c_i, c_j).$$

Each of the features captures one aspect of the synonymous ambiguities, and they all have their own limitations. Context-based similarity feature may fail to detect the ambiguous pairs from the same sentence, while pattern-based feature can mainly detect those from the same sentence. Textural similarity is only effective when the ambiguous pairs share common terms, and would fail to detect many that do not satisfy the requirement (e.g., the *account reference number* and its abbreviation *arn*). Thus, we propose the following method to combine all the features to improve the performance:

$$AS_S(c_i, c_j) = max\{Sim_P(c_i, c_j), (\alpha \cdot PS(Sim_C(c_i, c_j)) + (1 - \alpha) \cdot PS(Sim_T(c_i, c_j)))\}$$

where $PS(x)$ is the relative position score as shown in Equation (1). The proposed method trusts the results of pattern-based similarity more than other two features. When the two concepts do not follow any learned patterns, we will the consider their context and textual similarities. The importance between these two similarities is determined by the parameter α.

6 Experiment Setup

6.1 Experiment Design

Our system takes a set of SRSs as input, and then returns two separate ranking lists for the two kinds of ambiguities.

The pre-processing of the SRSs is kept to the minimum. We split the requirements into sentences, but did not remove stop words or stem the words. Stop words are not removed because they may be considered a stop word in one part of the document but used in a meaningful way in other parts of the document. For example, the words *to, be* are generally considered as stop words, but if these two words are removed, the concept *system to be* will lose its meaning. Word stemming is not used here because it may generate new ambiguity. For example, the concepts *programs, programmer, programming* are used correctly in the document without ambiguity. If word stemming is used, the three concepts will change to *program*, which could unnecessarily make the problem of overloaded ambiguity more difficult.

Results are evaluated with three measures, i.e., P@N (i.e., precision at top N results), R@N (i.e., recall at top N) and MAP@N (i.e., mean average precision at top

Table 1. Description of data sets

	Type	Domain	SRS length	# of Req	Req Length	# of Rev.
PI	Web-based software engineering tool	Software Engineering	7524	62	14	7
PII	Web-based business application	Business	5711	65	17	11
PIII	Web-based lending application	Banking	26823	272	14	16
PIV	Business application	Business	2294	60	29	17

N). P@N measures the percentage of top N detected concepts (or concept pairs) that are indeed ambiguous. R@N measures the percentage of ambiguous concept (or concept pairs) that are included in the top N results. MAP@N is a commonly used measure to evaluate the ranking results of top N results. Our primary evaluation measure is MAP@10.

6.2 Data Sets

We conduct experiments over four real-world data sets obtained from different software projects. These projects are chosen because they are real-world software projects, they span different domains and sizes, and there have been consistent efforts on revising the requirement documents. The characteristics of these projects are described in Table 1. The information includes the project name, project type, project domain, SRS length (in Terms), number of requirements, average requirement length and the number of revisions to the requirement documents for each project. The participants involved in **PI** and **PII** were software engineering students and professional developers with varying skills and experience, while those for **PIII** and **PIV** were professional developers.

To quantitatively evaluate the proposed approach, we create judgments on both ambiguity types for each project. Each judgment indicates whether a concept is overloaded ambiguous or whether a concept pair is synonymous ambiguous. The judgments are created by five assessors with training in software engineering and requirement engineering. For overloaded ambiguity, an assessor would go over all the candidate concepts for a project, and then decide whether each of them is ambiguous or not. The decision is made by first locating all the places where the concept was mentioned, and then check whether the concept has multiple meanings by reading the context of the concepts. The process for synonymous ambiguity is similar, while the assessor needs to compare the contexts of concept pairs.

The four projects were cross-evaluated by different assessors, for each project, there are at least 3 judgments for each type of ambiguity. With this judgments file, a voting schema is used to make the final decision. For each type of ambiguity of each project, we consider the candidate concept (pair of concepts) as ambiguous only if two or more assessors identified it is ambiguous.

Table 2 describes the basic statistics of the created judgments for each project. It includes the number of candidate concepts (i.e., *Concepts*), the number of overloaded concepts (i.e., *Overloaded*) and the number of synonymous concept pairs (i.e., *Synonymous*). It is surprising to see that a significant portion of the candidate concepts are still ambiguous even after at least 7 revisions, which reinforces the need for automated techniques that can help reduce these ambiguities and produce more consistent SRSs.

Table 2. Statistics of judgment sets

Projects	Concepts	Overloaded	Synonymous
PI	80	8	9
PII	66	23	3
PIII	143	11	7
PIV	57	7	6

7 Experiment Results

We now report the results for the proposed methods. There are several parameters in the proposed methods, so we train the parameter values on one collection (i.e., **PI**) and use the learned parameters for the remaining three test collections (i.e. **PII**, **PIII** and **PIV**). We conduct two sets of experiments to evaluate the effectiveness of the proposed methods for each ambiguity type, and report the optimal performance on the training set and the test performance on the testing sets for both sets.

7.1 Effectiveness of Overloaded Ambiguity Detection

Table 3 shows the optimal performance of the proposed overloaded ambiguity detection methods for **PI**. **All** denotes the method that combines all the features. **CDiv.**, **CFreq**, **NClusters**, and **InterDist** corresponds to the methods that use a single feature for ranking. They correspond to context diversity, concept frequency, the number of clusters in the context and inter-cluster distance respectively. During the training, we also conducted the 5 fold cross-validation on PI. The average MAP@10 measure of the proposed method (i.e., combining all feature) is 0.334. It is clear that combining all the features can consistently and significantly outperform the baseline method over all the test collections.

Table 3. Optimal Performance for *Overloaded* Detection on Training Set (PI)

Features	MAP@10	P@10	R@10
All	**0.42**	**0.5**	**0.63**
CDiv	0.24	0.3	0.38
CFreq	0.27	0.4	0.5
NClusters	0.21	0.3	0.38
InterDist	0.19	0.2	0.25

Table 4 shows the testing performance for the three test collections. Note that the parameters are set based on the values learned on the training set, i.e., PI. **All** still denotes the performance of combining all the features, and **BL** denotes the best performance when using a single feature. Moreover, the learned parameters on the training set seem to work well on the other test sets even if they are from completely different domains.

The similarity boundary is used as the stop criterion of the HAC method, i.e., when the maximum similarity value of two clusters is smaller than the similarity boundary, the clustering procedure stops. Therefore, the value of the similarity boundary affects the performance of *NCluster*, *InterDist* and *All* for overloaded ambiguity detection. We now examine the performance sensitivity with respect to the value of similarity boundary. Figure 2 shows the sensitivity curves for all the three methods on the training collection (i.e., PI). It is clear that the similarity boundary can not be either too large or too small. When the similarity boundary is too large, we may separate similar contexts

Table 4. Test Performance Comparison for *Overloaded* Detection

	All			BL		
	MAP@10	**P@10**	**R@10**	**MAP@10**	**P@10**	**R@10**
PII	**0.21**	**0.6**	**0.26**	0.11	0.5	0.22
PIII	**0.15**	**0.2**	**0.18**	0.08	0.1	0.09
PIV	**0.42**	**0.4**	**0.57**	0.12	0.1	0.14

into different groups. On the other hand, when the similarity boundary is too small, we may not be able to distinguish different contexts. For example, if the threshold is 0.1, most of the contexts will be grouped together and the ability to differentiate them is not limited. Our preliminary results suggest that the optimal value for the similarity boundary is around 0.3.

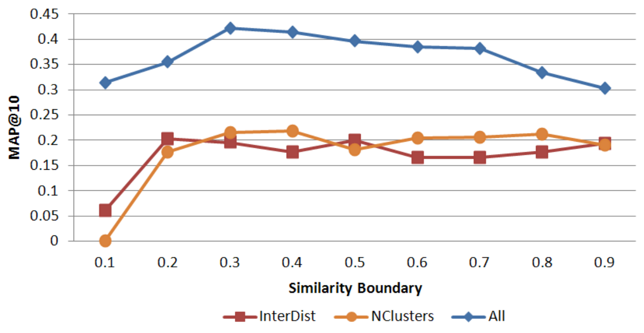

Fig. 2. Similarity boundary affects the performance(Project I).

7.2 Effectiveness of Synonymous Ambiguity Detection

We also evaluate the effectiveness of synonymous ambiguity detection methods. Table 5 shows the optimal performance on the training set **PI**. We conducted the 5-fold cross-validation for synonymous detection too. The average MAP@10 for using all features is 0.3. It is clear that using all the features is more effective than using individual features. In particular, using the textual-based feature outperform using the other two features. Furthermore, it is worth noticing that the context information is useful in detecting overloaded ambiguous concepts (all the features used in overloaded detection is based on context of the concept) but not helpful in detecting synonymous ones. The Contexts-based Similarity method does not perform as well as we expected. The reason, to our understanding, is because of the concept co-occurrence problem. Although currently the penalty is applied on the terms that show together, it is possible that two different concepts show in similar contexts but are not synonymous. On the other hand, it is not surprising to find that textual-based Similarity has a better performance, because similar concepts often share common terms.

With the parameters trained on Project I, we report the test performance on the other three collections in Table 6. **BL** denotes the baseline method using a single feature, and we use the textual based feature in this set of experiments since it is more effective than the other two features. Results show that it is more effective to combine all the features, and the conclusion holds for all the test sets.

Table 5. Optimal Performance for *Synonymous* Detection on Training Set (PI)

Feature	MAP@10	P@10	R@10
All	**0.31**	**0.4**	**0.44**
Textual-based	0.13	0.2	0.22
Context-based	0.07	0.2	0.33
Pattern-based	0.11	0.1	0.11

Table 6. Test Performance Comparison for *Synonymous* Detection

	All			BL		
	MAP@10	**P@10**	**R@10**	**MAP@10**	**P@10**	**R@10**
PII	**0.38**	**0.2**	**0.66**	0.16	0.1	0.33
PIII	**0.17**	**0.3**	**0.42**	0.09	0.3	0.42
PIV	**0.37**	**0.3**	**0.5**	0.13	0.2	0.33

We also conduct an exit survey with assessors and ask them about their experience in making the judgments for synonymous ambiguity detection. We find that it takes more efforts to make judgments for this ambiguity type, and it is necessary to consider both context and semantic meaning of the concepts to detect such ambiguities. Furthermore, the assessors also state that the ranked list is a good tool that can help them identify the ambiguous pairs more effective. In particular, the pairs remind them of some concepts that could be interchangeable, which was really helpful, especially when the SRS is long.

7.3 Discussions

Identifying ambiguous concepts from natural language is a difficult task, even for human assessors. To demonstrate that, we evaluated the judgment results from assessors. As every project have 3 sets of judgment, one of them is chosen as the golden standard to evaluate the remaining two. We iteratively conducted this evaluation in the project, and reported the average performance as shown in table 7. It is worth to notice that the performance of the manually created results is only around 0.5 for MAP. This low value proved that ambiguity detection is a challenging tasks even for well trained human assessors.

Table 7. Evaluation of manually created results

	Overloaded			Synonymy		
	MAP@10	**P@10**	**R@10**	**MAP@10**	**P@10**	**R@10**
PI	0.53	0.61	0.79	0.49	0.61	0.61
PII	0.49	0.78	0.49	0.46	0.68	0.76
PIII	0.47	0.48	0.50	0.36	0.61	0.38
PIV	0.52	0.51	0.53	0.57	0.58	0.67

8 Conclusions and Future Work

Our paper is one of the first papers that aim to detect ambiguous terminology from software requirements specifications. The problem is important yet under-studied. To tackle the challenge, we propose to formulate the problem as a ranking problem, and then discuss how to estimate the overloaded ambiguity scores for concepts and synonymous ambiguity scores for concept pairs. Experiment results over four real-world data sets show that the proposed combined methods are more effective than those methods using single features alone, and they have potential to help software engineers to detect ambiguity terminologies more efficiently.

Another interesting outcome from a software engineering perspective is the abundance of ambiguous terminology found in the four SRSs we used in the evaluation. The ambiguities were identified through a manual process, and averaged around 20% of concepts per SRS are ambiguous either because they are overloaded or synonymous. The large number of ambiguous concepts present, really reinforces the need for automated techniques that can help reduce these ambiguities and produce more consistent SRSs.

There are a few interesting directions for the future work. First, we plan to study how to automatically learn the weights for the proposed combined method based on the statistics of the data sets. Second, the detection performance is closely related to the quality of extracted concepts. We will study other concept extraction methods and see whether they are improve the detection performance. Finally, it would be interesting to study other types of ambiguities such as the scope ambiguity and attachment ambiguity [1, 2].

References

1. D, Berry.: Ambiguity in natural language requirements documents. In B. Paech, C. Martell. (eds.), Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs, LNCS, vol. 5320, pp. 1-7. Springer Heidelberg (2008)
2. D. M. Berry, E. Kamsties, M. M. Krieger: From contract drafting to software specification: Linguistic sources of ambiguity (2003). `http://se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf`
3. B. W. Boehm P. N. Papaccio.: Understanding and controlling software costs. In: IEEE Transaction of Software Engineering, vol. 14, pp. 1462–1477 (1988)
4. F. Chantree, B. Nuseibeh, A. de Roeck, A. Willis.: Identifying nocuous ambiguities in natural language requirements. In: Proceedings of the 14th IEEE International Requirements Engineering Conference, pp. 56-65, Washington, DC, USA (2006)
5. R. L. Cobleigh, G. S. Avrunin, L. A. Clarke.: User guidance for creating precise and accessible property specifications. In: ACM SIGSOFT 14th International Symposium on Foundations of Software Engineering, pp. 208-218 (2006)
6. C. Damas, B. Lambeau, P. Dupont, A. van Lamsweerde.: Generating annotated behavior models from end-user scenarios. IEEE Transaction of Software Engineering, vol. 31, pp. 1056-1073 (2005)
7. K. Frantzi, S. Ananiadou.: Extracting nested collocations. In: Proceedings of the 16th conference on Computational linguistics, vol. 1, pp. 41-46 (1996)
8. S. Greenspan, J. Mylopoulos, A. Borgida.: On formal requirements modeling languages: Rml revisited. In: Proceedings of the 16th international conference on Software engineering, pp 135-147, Los Alamitos, CA, USA (1994)
9. M. A. Hearst.: Automatic acquisition of hyponyms from large text corpora. In: Proceedings of the 14th conference on Computational linguistics, vol.2, pp. 539-545, Stroudsburg, PA, USA (1992)

10. I. Hussain, O. Ormandjieva, L. Kosseim: Automatic Quality Assessment of SRS Text by Means of a Decision-Tree-Based Text Classifier In: Seventh International Conference on Quality Software (QSIC), pp 209-218, 2007
11. N. Ide, J. Vronis.: Word sense disambiguation: The state of the art. In: Computational Linguistics, vol. 24, pp. 1-40 (1998)
12. C. D. Manning, P. Raghavan, H. Schtze.: Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA (2008)
13. C. D. Manning, H. Schütze.: Foundations of statistical natural language processing. MIT Press, Cambridge, MA, USA (1999)
14. D. Maynard, A. Funk, W. Peters.: Using lexico-syntactic ontology design patterns for ontology creation and population. In: Proceedings of WOP2009 collocated with ISWC2009, vol. 516 (2009)
15. A. Nikora, J. Hayes, E. Holbrook: Experiments in Automated Identification of Ambiguous Natural-Language Requirements. In: Proc. 21st IEEE International Symposium on Software Reliability Engineering, San Jose.
16. A. Porter and L. Votta.: Comparing detection methods for software requirements inspections: A replication using professional subjects. In: Empirical Software Engineering, vol.3, pp. 355-379 (1998)
17. A. A. Porter, L. G. Votta, Jr., V. R. Basili.: Comparing detection methods for software requirements inspections: A replicated experiment. In: IEEE Transaction of Software Engineering, vol. 21, pp. 563-575 (1995)
18. H. B. Reubenstein, R. C. Waters.: The requirements apprentice: an initial scenario. In: SIGSOFT Software Engineering Notes, vol. 14, pp. 211-218 (1989)
19. B. Roark, E. Charniak.: Noun-phrase co-occurrence statistics for semiautomatic semantic lexicon construction. In: Proceedings of the 17th international conference on Computational linguistics, vol. 2, pp. 1110-1116, Stroudsburg, PA, USA (1998)
20. F. Shull, I. Rus, V. Basili.: How perspective-based reading can improve requirements inspections. In: Computer, vol. 33, pp. 73-79 (2000)
21. S. Tratz, D. Hovy.: Disambiguation of preposition sense using linguistically motivated features. In: HLT-NAACL (Student Research Workshop and Doctoral Consortium), pp. 96-100 (2009)
22. A. Umber, I.S. Bajwa: Minimizing ambiguity in natural language software requirements specification. In: Digital Information Management (ICDIM), pp. 102-107 (2011)
23. A. van Lamsweerde.: Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley & Sons (2009)
24. X. Zhang, A. Fang.: An ATE system based on probabilistic relations between terms and syntactic functions. In 10th International Conference on Statistical Analysis of Textual Data - JADT10 (2010)
25. X. Zou, R. Settimi, J. Cleland-Huang.: Improving automated requirements trace retrieval: a study of term-based enhancement methods. In: Empirical Software Engineering, vol. 15, pp. 119-146 (2010)
26. D. Zowghi, V. Gervasi, A. McRae.: Using default reasoning to discover inconsistencies in natural language requirements. In: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference, , pp. 133-140, Washington, DC, USA (2001)