Parsing Natural-Language based Software Requirements into Code Segments

Tooraj Helmi
University of Southern California
thelmi@usc.edu

Abstract

In order to transform software requirements into code, we first need to identify and refine the existing ambiguities. In this paper, we introduce a systematic approach towards detecting ambiguities that depends on two key elements: a grammar that specifies the unambiguous syntax, and second, an NLP model to map software requirements to its terminal symbols.

1 Introduction

As shown in Figure 1, our research is closely related to two major disciplines: machine programming and ambiguity analysis in requirement engineering.

Machine programming is defined as using either rule-based (Lau et al., 2003) or probabilistic techniques (Balog et al., 2016) to turn a description of code into code. In the NLP community there has been a lot of research on converting code to comment. (Hu et al., 2018) for example uses NLP techniques to generate comments for Java code. However the inverse conversion, comment to code, is much more challenging due to structured nature of code where only a specific syntax is accepted by compilers. Due to this challenge, generated code by existing approaches does not necessarily compile. To resolve this problem, Some techniques like (Rabinovich et al., 2017) try to generate code based on an abstract syntax tree, however, these approaches only consider description that is completely clear and does not span beyond a few sentences. The description is either in shape of a few examples (Gulwani, 2016) or provided via natural language requirements to convey user's intention. As explained in (Gottschlich et al., 2018) intention along with invention and adaptation are three pillar of machine programming.

In requirement engineering community there is an area of research focused on detecting ambiguities in requirements. (Kamsties et al., 2001) identifies 5 different types of ambiguities that could exist in software requirement. Both heuristics based approaches (Yang et al., 2010) and NLP techniques (Dalpiaz et al., 2018) have been applied to detect these ambiguities. However, the existing techniques try to detect ambiguities in a vacuum, i.e., just by looking at specified requirements without considering the context which gives meaning to words and sentences used to express the requirements. This is probably due to the assumption that the context is normally known to a human programmer but clearly this is not the case when a machine tries to generate code.

When provided with requirements for a typical application, we do need to have some contextual information to interpret them – context being defined as entities, their attributes, actions that can take place on entities and qualifiers for either entities or actions. By relying on context defined in this manner, we can expand the definition of ambiguities beyond just the grammatical ambiguities. We categorize ambiguities into three types:

- **Isolated ambiguities** are introduced when a requirement is considered in isolation but can be clarified by information available in other requirements. For example. in requirement: "a [late] order should be cancelled", the bracketed term, late, is ambiguous; however, given another requirement: "An order is considered late if not delivered within 30 minutes", the ambiguity disappears.

- **Contextual ambiguities** are introduced by a missing context, when the term is not understood by the underlying processing system. For example, in "[order] should be [cancelled] if not [delivered] within 30 minutes", the bracketed terms are ambiguous in absence of a delivery context.

- **Grammatical ambiguities** are introduced in three different cases: first, when a term has more than one meaning (lexical); second, when there are multiple ways of attributing one part of speech to another (syntactic); and third when there is a complex phrase with a specific meaning (semantic). .

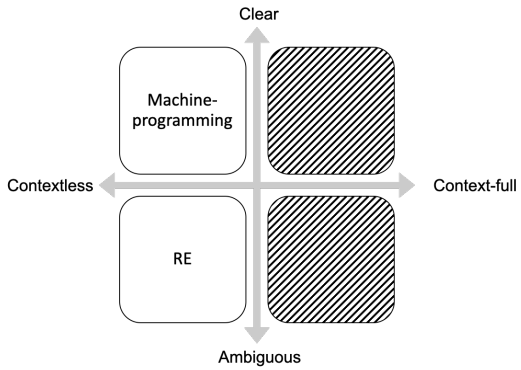

Figure 1: The hatched area on the right is neglected by both machine-programming and requirement engineering.

Figure 2 shows a simple view of the end-to-end process of converting requirements into code. It depends on extracting information from requirements, representing them in the form of a context graph and then using it to identify existing ambiguities in requirements and to generate code.

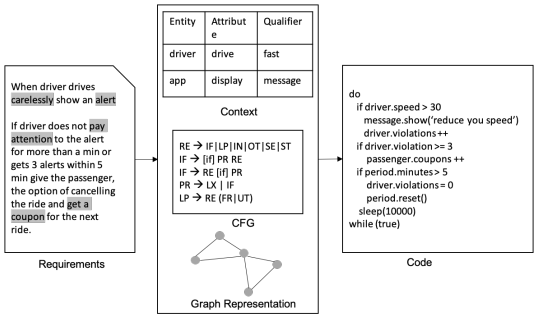

Figure 2: Transforming ambiguous natural-language based requirements to code requires three major elements: context, grammar, and representation

To achieve our ultimate goal of generating applications from requirements, we suggest the following 4 processes and tools:

1. A grammar and set of constituents used to extract key information from requirements specified in natural language.

2. A graph representation used to organize the requirements in the shape of a graph where adjacent nodes contain information about related entities.

3. A process to detect ambiguities in the requirements

4. A process to eliminate the ambiguities either automatically using the information already available in the requirements or by interacting with the user

In this paper we focus on the first item in the above list. The approach we use is to first define a grammar used to interpret requirements. We will then try to map requirements to the terminal symbols of this grammar.

2 Description of Problem

We try to address a specific problem: *How can we extract and interpret the information in requirements?*

This is a mapping problem from a set of plain requirements $\Re = \{r_1, r_2, ..., r_n\}$ to a grammar defined by $N \rightarrow (N \cup \Sigma)$, where N and Σ are non-terminal, and terminal symbols respectively.

An example can make our problem more clear. Consider a requirement stating that "If user picks the first choice, he should be shown a white balloon below the text". This requirement should be mapped to the following production rules: REQ $\rightarrow$ IFF, IFF $\rightarrow$ PRE REQ, PRE $\rightarrow$ LEX, and so on. Here REQ, IFF, PRE, and LEX denote requirement and conditional, predicate, and logical expression statements respectively and are some of the non-terminal symbols in our grammar. Note that each symbol has several definitions relying on other lower level terminal and non-terminal symbols. We can apply bottom-up deterministic or stochastic parsing algorithms like CYK (Kasami, 1966) or inside-outside (Lari and Young, 1990) algorithms to see if a mapping to the terminal symbols does exist. In this example, "picks" and "white balloon" will be eventually mapped to ACT and ENT terminals that denote action and entity respectively.

Defining a grammar that fits our purpose is not trivial. Competing criteria should be satisfied with this grammar: the grammar should be expressive enough to be able to generate majority of the desired applications yet it should not contain lots of terminal symbols so that we could apply NLP techniques to mapping the requirements to these symbol with a high performance.

#	Rule	Explanation
1	REQ → (IFF|LOP|INP|SHO) | (IFF|LOP|INP|SHO REQ)	Requirement
2	IFF → ([if] PRE REQ) | (REQ [if] PRE)	Conditional
3	PRE → LEX|IFF	Predicate
4	LOP → (REQ (FRE|UNT)) | (REQ (FRE|UNT) REQ) | ((REQ FRE)|UNT)	Loop
5	UNT → [until] PRE	Until
6	SET → [set] ENT [to] LEX|MEX|FIX	Set
7	FIX → NUM|TEX	Fixed value
8	OPE → FIX|STA|QLF	Operand
9	LEX → (OPE LOP OPE) | (ACT QLF)	Logical expression
10	MEX → OPE MOP OPE	Mathmatical Expression
11	STA → (ENT['S] ENT) | (ENT [of] ENT)	State
12	ENA → (ENT ACT) | (ENT ACT ENT) | (ACT ENT)	Entity-Action
13	INP → [enter] ENT | STA	Input
14	SHO → [show] ENT POS | STA	Show

Table 1: Context Free Grammar

Note that once we have a grammar we can use it to detect ambiguities: we can detect a syntactic ambiguity when at least one part of the requirement cannot be mapped to any production rule in the grammar. For parts that can be mapped to the grammar, we will then try to look for the words under the role of the terminal symbols assigned to them within a set of pre-defined context modules. These modules define what is known. If a match cannot be found, we mark it as a contextual ambiguity. The basic context that we call the APP context defines entities like user, app, menu, and actions like launching, showing, entering, navigating, and so on. In addition to the APP context, we can have domain and sub-domain specific contexts. For example a driving context can define entities like driver, speed and actions like driving, arriving and qualifiers like carelessly, fast, slow, and so on. In the previous example, "picks" and "white balloon" can be marked as contextual ambiguities if they are not defined in any of the provided context modules as action and entity.

3 Approach

We start with defining our grammar. We show how this grammar can be used to define applications and also point out its limitations. Once we have the grammar defined the next step would be finding a way to map the requirements to the terminal symbols. We will look at several NLP techniques for doing so.

3.1 Grammar

We have defined a grammar with 14 production rules shown in Table 1. Each of these production rules maps a non-terminal symbol to a set of either keywords or terminal symbols. We have designed this grammar to make sure it contains enough instructions to generate fairly simple applications that can interact with the user via a terminal or simple web or mobile apps that do not include complex visuals to take input, show output, and execute logical and mathematical expressions. Extending this grammar to support more complex applications will be an ongoing research.

Table 2 shows the non-terminal symbols for the grammar. We need to map these symbols to requirements for information extraction and ambiguity detection. We will go through the mapping process in the next section.

3.2 Annotated Dataset

We have created a dataset called Reqset, that contains requirements on different types of apps. Reqset contains around 4500 tagged words and can be used for any NLP research on software requirements. [1]. We have used a subset of non-terminal symbols specified in Table 2 to tag Reqset. These symbols include: ENT, ACT, QLF, POS, FIX, INP, SHO, ENT, and IFF. FIX is used to denote either NUM or TEX, and INT and SHO to denote [enter] and [show]. The reason we have not included all

[1]Reqset is available at https://toorajhelmi.github.io/home/publication/parsing

Symbol	Explanation
ENT	Entity
ACT	Action
QLF	Qualifier
POS	Position
MOP	Mathematical Operations: +, - ,...
LOP	Logical Operations: and, or , ...
TEX	Texts
NUM	Mumbers
[show]	Any verb that indicates showing something to the user
[enter]	Any verb that indicates taking an input from the user
[if]	Any word that indicates a conditional statement
[of] ['s]	Any proposition
[set]	Any verb that indicates setting an entity to a value
[to]	Used with set
[until]	Any word that indicates trigger of an event

Table 2: Non-terminal symbols

the symbols in Table 2 is due to two reasons: first, using more labels has an adverse effect on accuracy of our NER model. Second, once this basic set of tags are detected, we can rely on other techniques like dependency parsing (Chen and Manning, 2014) or semantic role labeling (Palmer et al., 2010) to extract other information.

Table 3 lists the apps are included in Reqset which contain 5 terminal apps, 3 desktop apps, and one mobile app. As can be observed from Figure 3, ENT and O (null) have the highest frequencies while IFF and INP have the lowest ones. This distribution is just a natural consequence of using English to describe software requirement where most of the words are nouns in English (Tsurukabuto) and conditional and input statements happen less frequently than other type of statements in programs.

3.3 Information Extraction

We have trained two different models on Reqset [2]. The first model shown in Figure 4 is a BI-LSTM, linear-chain CRF model that has been shown to perform very well on NER tasks (Ma and Hovy, 2016). The input sentences are padded to have a fixed length of 50. The first layer is an embedding layer that maps each word to a 20 dimensional vector. The second layer is a bi-directional LSTM layer with 50 cells that takes the embedded vectors one at a time and share their states with the neighbouring cells in both directions. This allows the network to remember the entire sequence. The last layer is a linear chain CRF layer (Lafferty et al., 2001) that considering all the previous input words and their predicated labels, generates probabilities for each of the existing labels.

[2]Code for these models is available at https://toorajhelmi.github.io/home/publication/parsing

App	Description
Trading	A Terminal app to buy and sell stocks
Tic Tac Toe	The terminal version of Tic Tac Toe game.
Word Guess	A terminal app to guess a word.
News	A desktop app to subscribe to see the news.
Food Delivery	A mobile app for ordering food.
Calendar	A desktop calendar app to manage events
Bank	A terminal app used by bank tellers.
Time Card	A terminal app to report time worked.
Alarm	A desktop app to set alarms.

Table 3: Non-terminal symbols

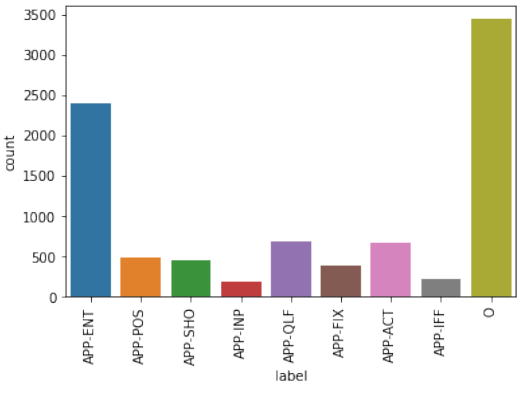

Figure 3: Distribution of tags on Reqset

We also fine-tuned BERT (Devlin et al., 2018) on Reqset. The reason for selecting this model was to potentially get a better result given the fairly small size of our dataset. To use this model we padded

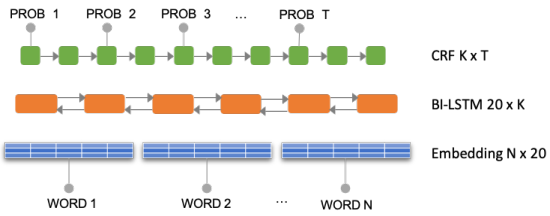

Figure 4: Model 1 is comprised of three layers: Embedding, BILSTM, and linear chain CRF

our sentences to have a fixed size of 50 words. BERT uses a special tokenizer which breaks the words if faced with words that do not exist in WordPiece (Schuster and Nakajima, 2012) vocabulary. In such cases, we used the original word's label for each of the generated pieces. However, to make sure it does not cause problems during prediction and scoring we applied the following correction mechanisms: First, if a word is broken to multiple pieces, we used the label with highest frequency among pieces as the word's label. Second, we used the reconstructed word label to calculate the test scores.

4 Experiments

We ran experiment to train our models on Reqest to predict each of our labels. We used requirements specific to "Time Card" and "Alarm" apps for testing and the rest of the requirements for training. This resulted in 3459-word training and 930-word testing sets. The training set was spitted 9 to 1 across training and validation. We trained our model with small batches of size 2 over 50 to 100 epochs. In order to measure the impact of adding more apps to Reqset we trained our models several times by incrementally adding requirements one app at a time. We saw an average increase of 0.27 on F1 score per each 100 words.

5 Analysis and Discussion

As shown in Table 4, both models provided comparable F1 scores, however, the first model resulted in a smaller standard deviation across F1 scores for individual labels.

Table 5 shows the performance of the models on a test sentence. We observe that model 1 (BILSTM-CRF) provides a much better accuracy (82%) compared to model 2 (BERT), although they both have the same macro and micro F1 scores. Such a significant difference across these models can be explained by the standard deviation on F1-scores each model produces for individual labels. As shown in Table 4 the standard deviation for model 1 and 2 are and 0.19 and 0.28 respectively. As shown in Figure 5, this means that model 1 provides a more consistent performance across different labels whereas model 2 does very good for some labels but does poorly for others.

Each model performs better for a specific set of labels. In general, model 1, provides a better performance on labels that depend on the syntax and model 2 provide a better performance on labels that depend on semantics. For example ACT and INP are normally a verb, ENT is normally a noun - either object or subject - and FIX is often surrounded by quotes. BILSTM and CRM components in model 1 are better equipped to detect such syntactic patterns and therefore offer a better performance for these labels. However, IFF and POS are normally assigned to words with similar meanings (if, when, whenever for IFF and below, above, to the left, ... for POS). BERT is better equipped to detect such semantic patterns and hence offers a better performance for these labels.

Some of the labels have been detected poorly by either model. This could be explained by the different usages of these labels. For example, QLF label is used to either qualify an action, e.g., driving [carelessly], or qualifying an entity, e.g., the [main] page. INP can refer to entering text using keyboard (enter, press, type) or tapping and clicking on non-terminal apps. On the other side, labels that are used on relatively defined patterns are detected much better. For example, IFF often comes at the beginning of the sentence and expressed using "if" synonyms.

The relatively good performance on ENT can be attributed to the frequency of having ENT in our dataset. As shown in Figure 3, it makes up around half of the labels.

6 Conclusion

In this work, we looked at using two tasks that can assist us with detecting ambiguities in require-

Model	Mi-F1	Ma-F1	STD
BI-LSTM-CRF	41	41	0.19
Fine-Tuned BERT	43	39	0.28

Table 4: Experiment Results. Mi-F1, and Ma-F1 denote micro and macro F1 scores. STD denotes the standard deviation across F1 scores for individual labels.

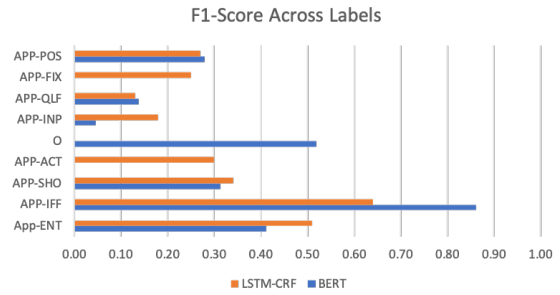

Figure 5: F1 scores that each model produces for different labels. Model 1 has a lower standard dev.

Word	True Tag	M1 Tag	M2 Tag
IF	IFF	IFF	IFF
USER	ENT	ENT	IFF
PICKS	ACT	QLF	ENT
THE	O	O	ENT
FIRST	ENT	ENT	O
CHOICE	ENT	ENT	ENT
,	O	O	O
HE	ENT	ENT	O
SHOULD	O	O	ENT
BE	O	O	ENT
SHOWN	SHO	SHO	ENT
A	O	O	O
WHITE	ENT	ENT	O
BALLOON	QLF	QLF	O
BELOW	POS	POS	O
THE	O	POS	O
TEXT	ENT	ENT	O
Accuracy		82%	35%

Table 5: Results on a benchmark requirement. M1 and M2 denote the BILSTM-CRF and BERT models. M2 does a much better job predicting the labels.

ments. First, we introduced a grammar to identify the interpretable structures and used it to decide whether a provided requirement is ambiguous or not; Second, we tried two different NLP models to map the terminal symbols of the grammar to requirements. The BILSTM-CRF model provided promising results.

7 Future Work

First, we would like to improve the performance of our NER. We are going to achieve this goal via different means, First, by expanding Reqset to include more requirements. Another way is to try other models including a combined BERT/CRF model that is equipped to consume both syntactic and semantic patterns. Another model we would like to build is what we call a "jagged NER" architecture. Jagged NER is going to be similar to Nested NER (Kamsties et al., 2001) but instead of having a rectangular shape, it will be jagged allowing us to detect different nested depths for different parts of the sentence. This could be beneficial since the nested segments in codes can have different depths. Third, we would like to include other features like syntactic and semantic dependencies in our modeling. Techniques like dependency parsing (Chen and Manning, 2014) and semantic role labeling (Palmer et al., 2010) can help us provide such features.

Second, once we can predict labels at a high level of accuracy, we will turn our focus on applying the grammar to detect ambiguities in the requirements and we will look at approaches that can help us refine those ambiguities. Our first attempt will be to build a context graph that can link requirements and help us extract relevant information from multiple requirements; second, by relying on techniques like knowledge graphs and advance language models like BERT (Devlin et al., 2018) or GPT (Brown et al., 2020) we might be able to automatically eliminate some of the ambiguities; Third, we are going to build a conversational AI system that can interact with the user to refine the remaining requirements.

8 Related Research

We review the existing research in three related areas: ambiguity detection in software requirements, NER, and incorporating context.

With respect to ambiguity detection in software requirements. In (Kamsties et al., 2001) authors divide ambiguities into 5 categories: lexical, systematic, referential, discourse, and domain classes. They offer a UML-based heuristic approach to detect some of these ambiguities. (Yang et al., 2010) categorizes ambiguities into nocuous - which occurs when text is interpreted differently by different readers - and innocuous - if different readers interpret it in the same way, even though structural or semantic hints exist. They develop a concept called ambiguity threshold and use it with a heuristic-based approach to detect the ambiguities. (Mishra and Sharma, 2019) tries to identify and detect domain-specific semantic ambiguities in natural language text. They apply an NLP technique based on word embedding to detect ambiguous words and show that word-embedding-based

techniques are very effective in identifying domain specific semantic ambiguities.

When it comes to NER, there are two areas of research that overlaps with our work. First area is around models that can detect nested relationships. Although we did not try to use a nested model in this paper, programming code is filled with nested structured and therefore we think using nested models can offer a better performance. (Ju et al., 2018) attempts to identify nested entities by dynamically stacking flat NER layers where each layer consist of BILSTM-CRF cells. They apply their models to a nested genetic dataset to detect nested relationships across genes, DNAs, and proteins. (Finkel and Manning, 2009) built a model using a discriminative constituency parser applied to detected nested relationships in news and biomedical datasets. The second area of overlap is based on detecting entities on mixed text and code media. (Tabassum et al., 2020) introduces a new corpus for the computer programming domain, consisting of 15,372 sentences annotated with 20 fine-grained entity types extracted from StackOverflow and trained a model based on BERT to detect 8 code-related and 12 natural language entities. (Ye et al., 2016) provides another StackOverflow based NER corpus.

Lastly, on incorporating contextual or external information in NLP, (Long et al., 2017) proposes two recurrent neural network architectures which make use of external knowledge in the form of entity descriptions and shows that the performance is significantly better at the rare entity prediction task when using external resources. (Liu and Singh, 2004) introduced ConceptNet knowledge base which is a semantic network consisting of over 1.6 million assertions of commonsense knowledge encompassing the spatial, physical, social, temporal, and psychological aspects of everyday life. Having access to such a knowledge base can assist us in removing some of the ambiguities without requiring an explicit context module.

8.1 References

References

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750.

Fabiano Dalpiaz, Ivor Van der Schalk, and Garm Lucassen. 2018. Pinpointing ambiguity and incompleteness in requirements engineering via information visualization and nlp. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 119–135. Springer.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Jenny Rose Finkel and Christopher D Manning. 2009. Nested named entity recognition. In *Proceedings of the 2009 conference on empirical methods in natural language processing*, pages 141–150.

Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. 2018. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 69–80.

Sumit Gulwani. 2016. Programming by examples. *Dependable Software Systems Engineering*, 45(137):3–15.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Meizhi Ju, Makoto Miwa, and Sophia Ananiadou. 2018. A neural layered model for nested named entity recognition. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1446–1459.

Erik Kamsties, Daniel M Berry, Barbara Paech, E Kamsties, DM Berry, and B Paech. 2001. Detecting ambiguities in requirements documents using inspections. In *Proceedings of the first workshop on inspection in software engineering (WISE'01)*, pages 68–80.

Tadao Kasami. 1966. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*.

John Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data.

Karim Lari and Steve J Young. 1990. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer speech & language*, 4(1):35–56.

Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156.

Hugo Liu and Push Singh. 2004. Conceptnet—a practical commonsense reasoning tool-kit. *BT technology journal*, 22(4):211–226.

Teng Long, Emmanuel Bengio, Ryan Lowe, Jackie Chi Kit Cheung, and Doina Precup. 2017. World knowledge for reading comprehension: Rare entity prediction with hierarchical lstms using external descriptions. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 825–834.

Xuezhe Ma and Eduard Hovy. 2016. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354*.

Siba Mishra and Arpit Sharma. 2019. On the use of word embeddings for identifying domain specific ambiguities in requirements. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 234–240. IEEE.

Martha Palmer, Daniel Gildea, and Nianwen Xue. 2010. Semantic role labeling. *Synthesis Lectures on Human Language Technologies*, 3(1):1–103.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE.

Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and named entity recognition in stackoverflow. *arXiv preprint arXiv:2005.01634*.

Nada Tsurukabuto. Noun/verb ratio in l1 japanese, l1 english, and l2 english: A corpus-based study.

Hui Yang, Alistair Willis, Anne De Roeck, and Bashar Nuseibeh. 2010. Automatic detection of nocuous coordination ambiguities in natural language requirements. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 53–62.

Deheng Ye, Zhenchang Xing, Chee Yong Foo, Zi Qun Ang, Jing Li, and Nachiket Kapre. 2016. Software-specific named entity recognition in software engineering social content. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 90–101. IEEE.