Implementing RFCs Using Natural Language Processing

Rashid Tahir & Justin Oswald
University of Illinois at Urbana Champaign

Abstract

The Internet Engineering Task Force publishes a formal specification of various internet standards in the form of RFCs. This creates a common understanding of these standards and provides a set of requirements essential for consistency and compatibility. However, the vagueness of natural language in these RFCs and the usage of customized programming constructs often results in implementations that are incompatible with each other. We propose a system that uses Natural Language Processing to translate RFCs into a semantically correct Intermediate Representation that can be verified and tested for functional correctness and ambiguity. We auto-generate source code from this IR using a system comprising four sub-components: a natural language parser, a database, a sticher and an automated debugger.

1 Introduction

The Internet Engineering Task Force (IETF) [1] is an organization dedicated to providing structure and consistency to today’s internet by producing technical documents with a detailed specification of how the internet should function. These documents, commonly known as RFCs (Request For Comments), are published with the aim to provide the research and industrial communities a global and uniform understanding of the various internet standards. To date, thousands of RFCs have been submitted to the IETF RFC repository [3].

RFCs play a pivotal role in standardizing internet protocols; however, they come with their fair share of challenges. To begin with, RFCs are long documents often spanning hundreds of pages, such as the Open Shortest Path First (OSPF) standard that is almost 250 pages long [46]. Furthermore, some standards are defined across multiple RFCs. For example, the Dynamic Delegation Discovery System (DDDS) is defined in RFC 3401, 3402, 3403, 3404 and 3405 [40, 42, 41, 39, 38]. Going back and forth between these pages can be quite tedious and time-consuming for a developer. Together, the length and nature of RFCs argue against manual implementation of these standards.

The ever emerging nature of the internet demands that protocols be constantly upgraded to keep up to date. Hence, the community updates these RFCs periodically, resolving errors and enhancing functionality, such as the transition of BGP-4 from RFC 1654 [53] to 1771 [54] and finally to 4271 [56] (with an update in RFC 6286 [15]). These updates necessitate the modification of legacy implementations, which is no mean feat given the mammoth size of the internet infrastructure.

At times, RFCs may be vague and unclear which is a direct consequence of the fact that they are expressed in natural languages. This creates ambiguity in the developer’s mind leading to a slightly different understanding of the same principal concept. Consequently, numerous implementations of common internet standards by various agents have emerged, for instance the implementation of BGP-4 by Cisco, NextHop Technologies, Alcatel and Laurel [24]. In fact, even when source code is available for developers to compare against, implementations vary substantially such as Quagga, which is timer based [57], compared to XORP which is event driven [23].

At some level, interoperability testing is performed. However, these implementations are not compliant with each other in their entirety [24]. This creates compatibility problems between Asynchronous Systems (ASes) resulting in reduced functionality and added complexity for troubleshooting procedures. For instance, if an event occurs in the network at the AS level, XORP will immediately update its state of the network as it is event driven. However Cisco (which is timer-based) will still be in the previous stage and modify its state only after the timer expires. During this period, both deployments will have an inconsistent view of each other which can result in packets being forwarded to routers that are no longer advertising a certain prefix [36]. It is, therefore, imperative that the internet community develop new ways of implementing these RFCs in a manner that enhances compatibility and interoperability.

Differences in interpretation of these RFCs are caught today through downstream compatibility testing on the various implementations produced out of the body of RFCs [24]. This is an expensive and time consuming process as correcting problems late in the implementation cycle involves significant rework and retesting. Even after all this testing has been done, it is still possible that implementation differences escape testing, creating compliance issues.
Clearly for large and complex standards, such as those produced by IETF, hand-coded implementations create errors and bugs based on conceptual differences between developers. Furthermore, vendors tend to overlook these differences if the base functionality is not affected to a large degree.

We propose a NLP System that would take as input the aforementioned RFCs and output source code by first reducing the RFC to a semantically correct intermediate representation. By converting the RFC into an intermediate representation we will provide a platform upon which language analysis and ambiguity reduction techniques can be applied. The system works by parsing each word in a sentence and placing meaningful tags on them. After the initial parsing and word tagging (entity plus part-of-speech tagging) is complete, the output will be fed into a language simplifier, intended to convert complex words into simpler words with the same meaning. Finally the simplified English can be fed into a semantic interpreter (language disambiguizer), which will output a representation of the RFC free of language ambiguity. This final IR will be verified for functional correctness and inserted into a system called Macho, responsible for generating source code based on the input.

Macho is a system that converts natural language queries into source code. Macho’s input is comprised of a description of the problem in natural language along with a test set of test cases. Together, the test cases and the description, forms a substantially concrete specification for Macho to interpret. Macho attempts to convert a sentence into a request for computation based on a mapping borrowed from the work of Biermann et al. It comprises four sub-components i.e. a language parser, a database of code, a code integrator and an automated debugger. We will modify Macho’s front end to provide it with the capability to process our IR.

All sub-components of the system, starting from the NLP parser to the code generator and the debugger, shall be fully automated. Such an automated approach will provide four primary benefits. Firstly, it will provide compliance with internet standards resulting in greater compatibility and conformance. Secondly, it will substantially reduce the complexity and cost of implementing these standards manually. Thirdly, it will provide the internet community a platform to test existing implementations for bugs and inconsistencies. Lastly, the system could potentially be used to highlight the vagueness found in these RFCs.

Our primary contribution will be the proposed NLP system based on four sub-components namely a parser, a word-tagger, a language simplifier and a semantic interpreter. Our system will output an ambiguity free intermediate representation which will be verified for functional correctness. Our secondary contribution will be the development of a query extraction module for Macho that will traverse our IR and extract a set of meaningful queries, which will be used to search for relevant classes in Macho’s database. This query extraction module will replace Macho’s existing parser.

2 Related Work

NLP: Natural Language Processing (NLP) is generally considered an AI-complete or AI-hard problem. The process involves solving four sub-problems: Named Entity (NE) recognition, Part-Of-Speech (POS) tagging, Word Sense Disambiguation (WSD), and lastly Parsing. Current research endeavors employ the use of Machine Learning (ML) algorithms and Statistical Analysis tools to develop heuristics that achieve reasonable performance in solving these sub-problems. Most of these strategies are limited to a particular domain such as monolingual dictionaries and perform poorly across domains. Furthermore they are often quite difficult to implement and require a significant training period in order to configure the ML algorithm for a particular data set.

Formalized Protocol Specification: Specification languages, such as the RAISE Specification Language (RSL) [20, 21, 47], SDL [9, 12], LOTOS [11, 65] and others [5, 30, 63, 22, 13] have been used in the past to specify various protocols and standards. However, the process of writing an entire specification in these languages can be quite challenging and protocol engineers are usually reluctant to embark on such a venture. Xiang et al. [67] show how the Alternating Bit (AB) protocol can be specified using RSL. They work with Finite State Machines (FSM) and Communicating Sequential Processes (CSP) models to present a comprehensive specification of the AB protocol.

Automatic Specification/Protocol Generation: Perrig et al. [51] present a novel mechanism called Automatic Protocol Generation (APG) that generates protocols automatically from the requirement specifications. The auto generation is followed by a screening phase where a protocol screener determines if the protocol is correct or not. APG was tested for two-party mutual authentication protocols and produced outputs that were both different and simpler as compared to ISO standard protocols documented in the literature. The authors advanced their initial work to three-party authentication and key agreement protocols. Similar to the APG system is the AGVI Toolkit [61] which finds near optimal protocols from system specification and goes one step ahead of APG by auto generating the source code in Java. The Controlled English to Logic Translation system (CELT) [58] works on a re-
stricted form of natural language and automatically translates it to First Order Logic (FOL) based on a large ontological framework known as Suggested Upper Merged Ontology (SUMO) [48, 4]. Substantial work has been done to auto generate access control policies and configuration files automatically from high level descriptions, such as the Firewall Management Toolkit [8] and others [18].

Automatic Code Generation: Previous attempts at automatic code generation, such as those in Matlab [52], B-Toolkit [31] and VDM++ Toolkit [26], work under the assumption that a formal specification, based on FSMs, CSPs or Matlab Algorithms etc, is available to the system which is used in the subsequent code generation phase. Micheal et al. [25] at NASA have presented an approach, which they call “Requirements to Design to Code” (R2D2C) in which protocol engineers write the entire protocol in constrained natural language. An Automatic Theorem Prover [28] infers the formal model, based on CSP, which is provably equivalent to the requirement specification. Finally the CSP model is converted to code either directly using tools such as FDR [62] or indirectly, from the B-Toolkit, by first translating to B [14]. PDIL [6] was a specification language specifically designed to specify, simulate, and automatically implement communication protocols in PARSEC.

3 NLP Evaluation

The first step towards our goal was the selection of a natural language processing library or the creation of one if no suitable existing project was found. To ease the integration process into the rest of the tooling, we constrain our final selection to Java based NLP systems. Other selection criteria includes accuracy, runtime performance (memory consumption and speed), open source, and output formats available. Given these constraints, we evaluated the Stanford Core NLP suite and the OpenNLP suite as main candidates. Additionally we tested the Berkeley Parser in order to have another comparison point.

3.1 Stanford Core NLP Suite

The Stanford NLP system, which utilizes the Penn Treebank 3 tokenizer, includes 3 pre-trained models with the bundle. Each model averaged between 1 and 2 incorrect tags per sentence. As a result across the 3 pre-trained models, our sample data had a POS tagging accuracy between 92.32% and 96.1%.

3.2 Berkeley Parser

The Berkeley Parser, which also produces as output an unspecified variant of the Penn Treebank format featured similar performance characteristics to the Stanford Core NLP system. One intriguing feature of the parser is its wide variety of output formats, from graphical to flat text. Additionally it features the capability to generate the k best POS trees for some input, enabling downstream processes to select from multiple interpretations in the hopes of finding one that best suits the needs.

3.3 OpenNLP Suite

OpenNLP suite, utilizing the Penn Treebank 2 tag set included one trained POS model, the en-pos-maxent model, trained with a tag dictionary. The average number of wrong tags per sentence was 1.3 giving a POS tagging accuracy of 95% on our test data.

3.4 NLP Selection Summary

In general we found all three implementations to be quite similar in terms of POS tagging accuracy. In some instances we find that the result is more specific than needed for our purposes; the impact of such overspecification is still unclear. For instance, in the example sentence “The current set of next hops was calculated.” the word “hops” was tagged as a verb, 3rd person singular present. Though the description of 3rd person does not apply, and it would have been more than sufficient to tag the word as a verb alone, the important part is its identification as a verb for processing in the rest of the system. Further experimentation will reveal how, if at all, these extra identifiers will affect the final system.

To focus the research, we choose the Stanford Core NLP Suite as the processor of choice for our system. Though OpenNLP contains superior documentation and in our experiences, OpenNLP is the simplest system to integrate into a Java based program, the Stanford system has rich support for lemmatisation. The inclusion of the lemmatisation components will easily offset the poor documentation penalty that comes with using the Stanford system.

Stanford Core NLP Suite is also an open source solution, with a large list of accompanying libraries that will be useful in constructing a useful RFC processor.
4 System Design

The high level design of our system is seen in Figure 1. Most of the processing is done in two main modules, the Natural Language Processing Module and Macho, which is further comprised of several subcomponents. An Intermediate Representation (IR) is used to transport processed information between the two primary modules. Other components of our system either perform verification or highlight ambiguous portions of RFC text.

4.1 Natural Language Processing Module

The first module is the natural language processor which takes RFC text as input and converts it into an annotated form easily processed by a machine. Based on the evaluation work done in Section 3, we use the Stanford tool suite to function as our NLP processor. Out of the box, Stanford NLP splits the sentence into Penn Treebank 3 (PTB3) tokens and performs Parts Of-Speech (POS) tagging, again based on the Penn Treebank POS Tag Set, on the input text and produces a rudimentary Penn Treebank output.

In order to clean the input text, we add several layers of processing to reduce language and sentence complexity as shown in Figure 2. After running the text through the POS tagger, we run the named entity recognizer. This process creates customized domain specific categories which will be useful during the ambiguity recognition phase. Next we perform lemmatisation. Lemmatisation is a process where different forms of a verb are reduced to its simplest form by extracting the basic underlying lemma. This process reduces the complexity of the input sentence, making term matching easier in the downstream processes. The last subcomponent of the NLP Module is the parser, which produces an intermediate representation (Section 4.2), based on the information collected in the previous phases, to be used by the rest of the toolchain.

4.2 Intermediate Representation

In order to facilitate communication between the NLP front end system and Macho’s Query Creation Module, some intermediate representation of the language is needed. Additionally, this intermediate representation must be suitable to perform sentence evaluation, analysis, and simplification as discussed in Section 4.1. As the standard output of all the NLP systems evaluated is Penn Treebank, we consider this format to be a standard for NLP systems. It meets the needs of both the NLP components as well as being a suitable transfer mechanism to the Query Creation Module.

4.3 Ambiguity Recognizer

The Penn Treebank IR is passed on to the Ambiguity Recognizer. This module also takes as input the original RFC text. It identifies the ambiguity in the IR and then highlights the corresponding sentences in the original RFC text.
In our particular case, a sentence is identified as being ambiguous if either the parser fails to parse it or if the parse score is below a threshold value \( T \). Details on how the parse score is calculated can be found in the Stanford Parser Specification [29]. Broadly it is a per sentence measure of how accurately and successfully the parser is able to parse a particular sentence. The value of parameter \( T \) is determined empirically and will be different for different domains. We found that in our particular case, a parse score below 200 was often indicative of some sort of ambiguity in a sentence and therefore we set this as the threshold value during our experiments.

### 4.4 Query Extraction Module

The Query Extraction Module takes as input a simplified Penn Treebank text from the NLP module, extracting nouns and verb relationships in order to form queries. As demonstrated in Figure 3, the set of queries is executed against a massive database of Java source code by Macho in order to find the best candidate class files that will be stitched together in the Code Integrator Module. These candidates will be passed to the Automatic Debugger which derives a working program meeting the requirements of the RFC. For a comprehensive explanation of Macho and its working we refer the reader to [7].

### 5 Training

The Stanford Parser can be trained over annotated data to improve the performance of the parser on domain specific text. In our particular case, we wanted to train the parser so that it could parse RFCs with greater accuracy. As mentioned in Section 4.3 if the parser fails to parse a certain sentence, or if the parse score for that sentence is below a certain threshold, we categorize that sentence as ambiguous. Working with the standard parser gave us too many false positives. In order to reduce the number of false positives generated by our scheme we had to train the parser over RFCs.

The process of training the parser proved to be quite difficult and time consuming. In order to train the parser, RFC text was manually annotated with Penn Treebank tags. 723 sentences were used from RFC 1771 [53], which is the obsolete specification for BGP. The training data had 18,027 words in total, each of which had to be labeled using the Penn Treebank notation.

We trained the parser in two separate ways. The first technique we adopted was automated annotation where we trained the parser over annotated data derived from the parser itself. In this mode of training we fed the parser our training data and obtained an annotated document at the output. This technique was quite simple and allowed us to work with a larger training data. However, the resulting Treebank lacked accuracy as the parser itself annotated the data without any prior training.

The second mode of training involved the manual annotation of RFC 1771. We traversed the document in a simple text editor and labeled each word according to the Penn Treebank tag set. The resulting Treebank had greater accuracy and served as better training data. Since the process was quite time consuming we could only annotate a fraction of the total document.

We present the results in section 6 that highlight the effects of training the parser.

### 6 Results

Results of training method 1, where RFC output text was fed back in as training data to the parser, is shown in Figure 4. Obviously, the more lines of training data given to the model trainer proved to have a direct correlation with the number of correctly parsed sentences.

It can be seen that there is a drop to 0 in number of correctly parsed sentences on our test data between 4,000 and 10,000. As this is a statistical based parser, we suspect that the training data added in these lines was too different from the other sentences in the training data such that the trained model was not effective. As the size of the training data increases anomalies such as the one shown here should disappear.

The second method for training the parser involved manually annotating training text with Penn Treebank tags. The results of this exercise are shown in Figure 5. Due to the length of time required to hand annotate the input
text, the training data was a much smaller size and therefore results exist for up to 4,000 lines only.

The manually trained system shows an increase in the number of correctly parsed test data sentences as the number of lines of training data increases. Though there is a small dip in the number of correctly parsed sentences we believe again this is a symptom of the system being a statistical based parser without sufficient training data. We expect these minor variations to disappear as the size of the training data increases.

When one compares Figure 4 to Figure 5 it is clear that the accuracy of the parser is much higher when manually trained. We conclude that standard trained models bundled with parsers do not have the same degree of accuracy as parsers trained on domain specific text.

7 Future Work

During the course of this project we encountered several issues with processing RFCs that must be addressed in future work.

Often times RFCs include graphics, drawings, or technical diagrams that clarify or enhance the idea being conveyed. Surrounding text may reference a figure or diagram to augment the ideas being presented in natural language. As NLP systems are designed to work with natural language only, currently these sections must be manually excluded from the input to the system. In the future, this information should be converted into usable textual descriptions in order to add to the systems understanding of the text. One technique that can be applied is a domain specific image recognizer. For example, when processing computer science documents, an image scanner which understands standard UML symbols should be utilized to extract UML information from the document. Another possibility is to use optical character recognition to convert text in images to plain text.

Related to image references, chapter or section references must currently be discarded as they are meaningless to the NLP parser. References in RFCs and other technical writings serve as a link between related parts of a document. If this linked information, or an extract from the linked target is included in the parsed output, a greater understanding of the system and relationship between components will be achieved.

Currently our system trains only the POS tagger component of the parser. Additionally training the named entity recognizer will improve the ability of this module to recognize relevant terms (classes or objects) to improve the comprehension of the text.

As Macho, in its current form, operates on a database of Java classes, it does not take code comments into account while searching the database. When descriptive and up to date, comments may provide an easy indicator to the intent of the code. This can serve to prune Macho’s search space to more relevant assemblies of classes. Additionally, in some instances, a programmer may refer to a requirement or other text in an RFC, which can improve Macho’s matching results.

Lastly, another technique to improve results is to run multiple NLP parsers or parser the same text with different trained models over the same input text, thus allowing the ambiguity checker to select the least ambiguous result to pass onto Macho.

8 Conclusion

Software specifications and standards are in a constant state of flux in today’s fast paced world of software development. Validation of standards proves difficult as there
may be a large group of teams working on the standards with poor coordination. New techniques are required to improve the validation and realization of these standards as they are continuously undergoing enhancements, updates, and clarifications.

By uniting natural language processing with Macho, an automated program creation system, we produced a system able to comprehend the natural language found in the standards documents and generate a program to implement the standard given a library of classes to choose from. This system promises to improve the time it takes to prototype standards changes and improve the reliability of the documents produced.

References


