CSI6900- Graduate Research Project – Fall 2020

Submitted by

Hariprasad Ramakrishnan (300095834)
Master of Computer Science – hrama039@uottawa.ca

Under the guidance of

Professor Dr. Mehrdad Sabetzadeh
School of Electrical Engineering and Computer Science
University of Ottawa
800 King Edward Avenue,
Ottawa, Ontario, Canada, K1N 6N5.
m.sabetzadeh@uottawa.ca

Automatic Translation of Natural Language Requirements into Logic

Hariprasad Ramakrishnan
School of Electrical Engineering and Computer Science
The University of Ottawa,
Ottawa, Canada
hrama039@uottawa.ca

***Abstract*— The automatic translation of natural language requirements into logic is closely related to the area of requirement extraction and can be considered a requirement engineering problem. As part of this work, we will be working on natural language data, which is a part of the requirement specifications of a satellite control system. The following are the high-level tasks associated with the research work. The main aim would be to identify and extract various useful components from the natural language text using multiple natural language processing approaches. We would perform data cleaning, pattern identification, pattern-based grouping, and data pre-processing before the extraction. Post extraction of components, we would be arranging them into a format that would closely resemble structural language. The evaluation will be done manually, and the observations will be recorded systematically.**

Keywords—natural language, requirements, parsing, components

I. INTRODUCTION

Requirements are the basis for any software that drives today's world. The discipline of defining, organizing, and maintaining requirements falls under the broad area of requirement engineering. Requirement Engineering [1] is an essential step in the software development process because high-quality requirements provide developers with a precise understanding of what a proposed system needs to do to meet its stakeholders' needs. Most requirements are captured in natural languages such as English. Requirements captured in natural language might be easy for humans to comprehend but are not readily analyzable by computers. In many cases, requirements are manually read by users and processed into information like states, transitions, conditions, variables, which might then be useful in the further phases of development.

The volume of requirements that need to be manually read and processed is high, and the requirements are not structured. [2] This makes the task of manually extracting structured information from natural language documents time-consuming, labor-intensive, and prone to human-fatigue errors. Hence, there is a need for Natural Language Processing systems that convert unstructured natural-language requirements into structured artifacts or some formal logic that can be interpreted by machines more efficiently than unstructured natural language.

As part of this project report, we would see the various steps we followed in the project's research and analysis. Section 2 contains the background and the related works where we would discuss the motivation behind this research also discuss the problem description in detail. We would also have a look at similar work(s) that were doing in this field. Section 3 would talk about the technical approach, where we would go through the various tools and technology choices and the reason for choosing it, including its pros and cons. After this, we will brief the environment setup at a high level, the NLP library concepts that we will use, and the high-level design of the project's stack. Section 4 talks about implementing the project, which starts with the data set formation, data cleaning, and pattern grouping. Section 4 will continue discussing the various components that can be extracted and the methodology used to extract them. Section 5 will cover the empirical evaluation, where we examine the testing approach and discuss the various observations. Section 6 will explain the results, observations, and discussion in detail. Section 7 will contain the lessons learned while performing this project or research. Section 8 will include the threats to our research's validity, and section 9 will consist of the research conclusion.

II. BACKGROUND AND RELATED WORK

A. Requirement Engineering and information extraction

As our core problem lies in the area of Software Engineering, we would need to get a good idea of the kind of approaches and research works that are carried out in the software engineering domain and requirement engineering more specifically. The research [3] of Heitmeyer, C. et al. give us a good understanding of consistency checking, which is a problem related to requirement engineering, and how they have approached the problem. The work also provides us with an idea of the various software engineering models that can be used to manage the extracted information.

B. Motivation

Our project's base paper is the work [4] done by Menghi, C., where they generated automated test oracles for Simulink models. One step prior to their work involves writing various first-order logic requirements extracted from Natural language requirement specifications. This process was done manually, consuming a lot of time. Our research aims to automate this manual task and automatically translate requirements in natural language into logic. A sample snippet of the text data and its translated logic form can be seen in figure 1. Our research will aim to extract the info and assemble them in a structured language that can be refined and processed by programs without any human intervention.

C. Extracting Components from Natural Language Requirements

Various components need to be extracted to represent a system in the form of a notation. There are multiple notations in which we can use our extracted data and describe the system. The state-machine diagram, activity diagram, and many more are

ID	Requirement	Restricted Signal First-Order logic formula*
R1	The angular velocity of the satellite shall always be lower than $1.5m/s$.	$\forall t \in [0, 86\,400): \|\vec{w}_{sat}(t)\| < 1.5$
R2	The estimated attitude of the satellite shall be always equal to 1.	$\forall t \in [0, 86\,400): \|\vec{q}_{estimate}(t)\| = 1$
R3	The maximum reaction torque must be equal to $0.015Nm$.	$\forall t \in [0, 86\,400): \|\vec{trq}(t)\| \leq 0.015$

Figure 1 Text requirement and its manually converted logic from base paper

a few of the many notations that would give us an idea of what all components need to be extracted from the requirements to represent the system.

The work done by Muslea I. et al. [5] provides us with an excellent understanding of the various tools, techniques, and methodologies. We got an idea that our problem can be encountered by using rule-based and pattern-based extraction methods. When researching about the various components that can be extracted and their extraction methodology, the research work done by Pudlitz, F. et al. [6] gives an idea regarding how system states can be identified and extracted from natural language requirements. They use an automatic and semi-automatic method, but they use LSTM Deep learning models and Word Embedding to perform the extraction. In our case, we would not be using any Machine Learning models explicitly but will only use pattern and rule-based techniques.

On reading the work done by Allan, J. et al. [7], we can understand how they extracted event data from natural language requirements. The above gives us pointers on what approach we can use to detect events and conditions. Analyzing and reading about the work done by Zhou, Y. [8] provides us with an idea on how to extract the comparators using the dependency parsing concept in NLP. They extract the comparator phrases from the methods comments of Java docs. We will be using dependency parsing based techniques in our work to extract various components.

III. Technical Approach

In this section, we will be discussing the technical side of the research. We will be discussing the design choices, available tools and technology that can be used to perform the tasks, the environment set up, the development process followed, and some key concepts that form the core of the implementation.

A. Tools and technology choices

There are various NLP tools and technologies that are available for use. Some of them are free and open-source, while some of them are not. Since this is an academic project, we decided to go with free tools and technologies. NLTK, Gate, DK-Pro, and Stanford Core NLP were the various choices that were shortlisted. Our project's main criteria are that the library or tool, or technology should perform syntactic and semantic parsing accurately. We found that NLTK does not offer very accurate syntactic or semantic parsing compared to Stanford Core NLP on performing some proof-of-concept tests.

The tests were made purely on randomly chosen data and empirical observations. The tests were conducted just to do a quick validation of the libraries, and hence they were not recorded formally. Additionally, Stanford Core NLP had the concepts of Tregex and Semgrex, which seemed to be very useful for our problem. This will be discussed in detail later. Since Stanford Core NLP had utilities like Tregex and Semgrex, which are very useful, we chose Stanford Core NLP as our primary choice of the library.

B. Setting up Analysis environment

To analyze the data manually, we used a public API of Stanford Core NLP, which can be accessed via the internet at the URL: https://corenlp.run/. However, if you wish to access it locally on a server on your desktop, it can be done by downloading the Core NLP package and running following the Stanford Core NLP official website [9].

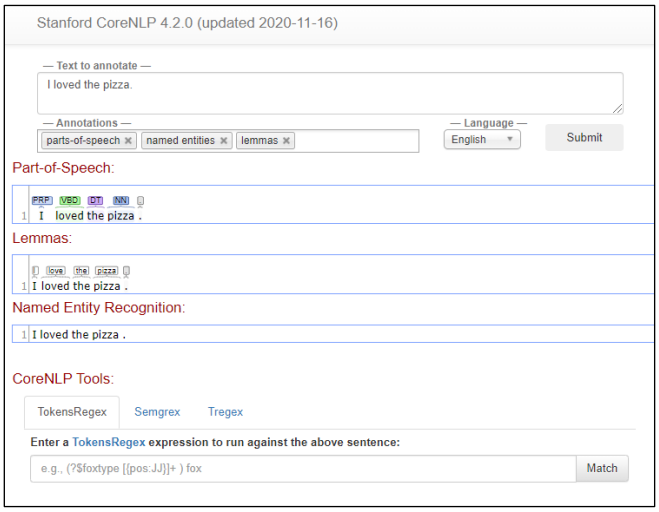

Figure 2 Stanford Core NLP User Interface

C. Setting up the primary project environment

We designed it to be a Java Maven project. To use the Stanford Core NLP Library, we only need to add the respective dependencies as part of the pom.xml file of the maven project. A snippet of the pom.xml file of our project is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apac
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>stanfordnlp</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>edu.stanford.nlp</groupId>
            <artifactId>stanford-corenlp</artifactId>
            <version>4.0.0</version>
        </dependency>
        <dependency>
            <groupId>edu.stanford.nlp</groupId>
            <artifactId>stanford-corenlp</artifactId>
            <version>4.0.0</version>
            <classifier>models</classifier>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.12</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.opencsv</groupId>
            <artifactId>opencsv</artifactId>
            <version>4.1</version>
        </dependency>
    </dependencies>
</project>
```

Figure 3 Maven Dependency management

Recent releases of Core NLP need at least Java 1.8+ as a pre-requisite. As you can see, we have used the 4.0 version of Core NLP here. The highlighted dependencies are all that we need to make Core NLP up and running. Apart from that, Project Lombok is a handy library that helps to avoid writing verbose POJO (Plain Old Java Object) classes as its annotations simplify various verbose java code. OpenCSV is the library that we have used to read and write CSV files. If we use a framework like GATE or Dk-Pro, we will find some out of the box options to read from and write into CSV files. As the Pom.xml file is framed like above and saved, maven will automatically download all the Stanford Core NLP related dependencies into your project's library.

D. Development process and repository info

The project is available as part of a git hub repository [10]. This report, the raw data set, and the cleaned data set are also available as part of it. Just like any modern development process, this project also followed distributed version control.

E. Stanford Core NLP concepts

Let us look at the fundamental concepts of Stanford Core NLP [9] in brief. Core NLP is written in Java and is a very rigid and robust framework. It supports six other languages apart from English

1) Pipeline: A pipeline is the central idea of any NLP implementation. A pipeline takes in raw text, performs a series of annotations on it, and finally renders an annotated document. For our project, we have a 'pipeline' java class that has been designed to generate a 'singleton' object. This way, even if we are processing multiple sentences, one pipeline object will be created and used to process all the text instances. This approach saves a lot of time and memory.

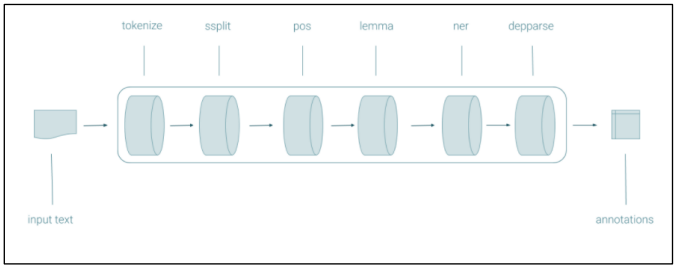

Figure 4 Sample Core NLP Pipeline (Shared from Core NLP [9] documentation)

2) CoreDocument: The output of a pipeline is a CoreDocument. It contains annotated data. This has various utilities and can be used to perform a lot of NLP tasks with ease. Therefore, it will contain the sentences with annotated tokens.

We can extract the sentences and tokens easily by using the inbuilt functions of the CoreDocument class. CoreDocument can also be the input format for our pipeline. Once the pipeline gets executed, it is also possible to extract only the required annotation using specific methods. Each token in a CoreDocument is called a CoreLabel. For instance, the method coredocument.tokens() returns the list of CoreLabel tokens.

Source code repository: https://github.com/hariprasadr92/NatLangToLogic

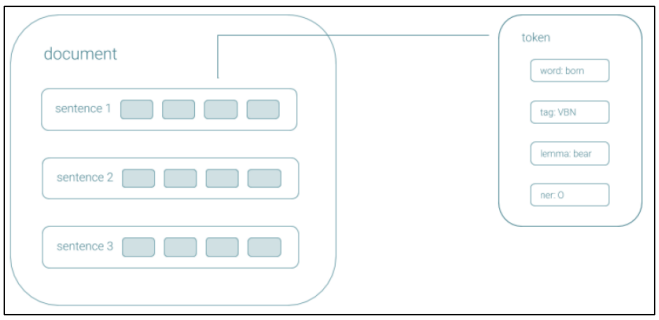

Figure 5 Core document in Stanford Core NLP (Shared from Core NLP [9] documentation)

3) Annotations: Annotations are like meta-information that is associated with each token. Stanford Core NLP has different annotators, including parts of speech, named entities, dependency parse, coreference, and many more.

4) Constituency parsing: This is the syntactic parser or the chunker. The Stanford Core NLP constituency parser breaks down the sentence into constituents or chunks. The output of a constituency parsing is a constituency tree, which we will be using in our implementation

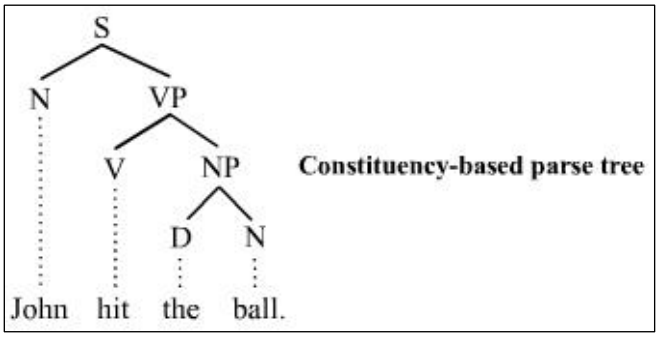

Figure 6 Constituency parsing illustration

Figure 7 Constituency parsing example

5) Tregex: Tregex [11] is the utility provided by Stanford Core NLP to perform a regex like operation on the constituency tree. We will be using tregex in few places in our implementation. After we perform tregex on a tree, the output will be a list of several subtrees that match the given tregex expression.

6) Dependency parsing: This is the semantic parser of the Stanford Core NLP. The dependency parsing output is a Dependency graph with edges with two nodes representing a governor node and another depicting a dependent node. The edge represents the relation between them or, in other words,

the dependency from the governor to dependent. As part of the implementation, we would be using many dependency parsing to extract various components from the requirement. In Figure 8, for the *nsubj* relation, the verb 'prefer' is the governor, and the subject 'I' serves as the dependent.

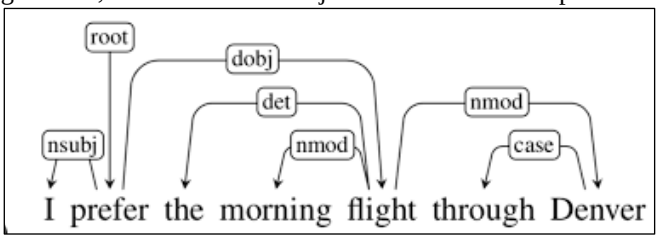

Figure 8 Dependency parsing illustration

Figure 9 Dependency parsing example

7) Semgrex: Semgrex [12] is similar to tregex, but it performs a regex like operation on the dependency graph. The output of Semgrex is the list of nodes that satisfy the semgrex expression. Please note that Semgrex is applied on a dependency graph and not a tree.

IV. Implementation

In this section, we will be discussing the implementation approach in detail. We will start with extraction of CSV from a flat file, data cleaning, pre-processing, and components extraction.

A. Data Extraction

One of the authors of our base paper [4] helped us secure the requirements data, which was a flat file. This file had to be converted into a CSV format. This initial data extraction was done manually. Please find before and after snapshots below.

```
Oracle
ModelName: Lux_Model;
Path: ./Lux_Model/;
ID: Lux_Req_1_1;
Signals w_sat;
Constants T=86400;
Description: "
SatEx model:
A satellite control system which helps always keep the satellite on the desired orbit,
ensures that the satellite is always facing the earth, and regulates the satellite speed.
Requirement:
The angular velocity of the satellite should always be lower than 1.5m/s,
i.e., the modulus of the satellite angular velocity w must be less or equal to 1.5m/s.";
Specification:
(
        forall t in [ 2700 , T ]:
         (( ||w_sat(t)||  < 1.5 ))
);

Oracle
ModelName: Lux_Model;
Path: ./Lux_Model/;
ID: Lux_Req_1_2;
Signals q_estimate;
Constants T=86400;
Description: "
SatEx model:
A satellite control system which helps always keep the satellite on the desired orbit,
ensures that the satellite is always facing the earth, and regulates the satellite speed.
Requirement:
```

Figure 10 Raw Data File

	Model_name	Path	ID	Signals	Constants	Descripton_model_	Descripton	Requirement
2	Lux_Model	./Lux_Mo	Lux_Req_	w_sat	T=86400	SatEx model	A satellite control	The angular velocity of the satellite should
3	Lux_Model	./Lux_Mo	Lux_Req_	q_estimati	T=86400	SatEx model	A satellite control	The estimated altitude of the satellite
4	Lux_Model	./Lux_Mo	Lux_Req_	torque	T=86400	SatEx model	A satellite control	the maximum reaction torque (torque)
5	Lux_Model	./Lux_Mo	Lux_Req_	q_error,q_	T=86400	SatEx model	A satellite control	The satellite should reach the desired
6	Lux_Model	./Lux_Mo	Lux_Req_	q_real,q_e	T=86400	SatEx model	A satellite control	The satellite should estimates its position
7	Lux_Model	./Lux_Mo	Lux_Req_	sm,q_real,	T=86400	SatEx model	A satellite control	After teh satellite enters the safe pin
8	Lux_Model	./Lux_Mo	Lux_Req_	KE,CE,PE	T=86400,	SatEx model	A satellite control	The satellite shall constantly keep its
9	Lux_Model	./Lux_Mo	Lux_Req_	rw_h	T=86400	SatEx model	A satellite control	The satellite reation wheels shall never be
10	Autopilot_Model	./Autopil	Autopilot	APEngAilC	T=200000	Autopilot model	A full six degree of	When the pilot selects the autopilot

Figure 11 CSV Data File

B. Data Analysis and Pattern-based grouping of data set

The next step would be to analyze and understand the data. As part of this step, we would manually parse the data using the Stanford Core NLP endpoint to identify various patterns. As part of this work, we identified and grouped the sentence into eight different categories.

As part of data analysis, we identified the requirement to have two main components: Action and Condition. Actions are events, and they might or might not have trigger points. It can be a simple event, transition, or a rule. An action can be further broken down into sub-components like subject, verb, comparator, object, value, and units. An action may or may not have a pre-condition. For example, Type 1 to 4 do not have any pre-conditions, and type 5, 6, and 7 have some conditions. There can also be more

Type	Buckets
1	Action with clear Subject Comparator and Value
2	Action - Subject and Vakue can be extracted. Comparator is not clear
3	Action- Only Subject can be identified
4	Action - Same as 1 but object is not a value but another subject
5	Condition triggers Transition type action
6	condition triggers events
7	multiple conditions triggers series of events

Figure 12 Categorization or bucketing of the dataset

than one condition and more than one action. For instance, type 7 has multiple triggers (conditions) and causes various events. Some events can be transitional, which means they move an object from one state to another state. These events are called transitional events. There are a set of sentences, which are too complicated to undergo the component extraction process. They are grouped as type 9. The categorization using pattern is only to understand the data better and aid in the extraction process. There are no separate code logic based on what kind of category or type the requirement belongs to. The categorization also helps in testing when we manually evaluate our results category wise.

C. Data cleaning and Data pre-processing

NLP pipeline depends on the quality of the data provided to it. The natural language dataset should not have errors like grammatical errors, typographical errors, and spelling errors. In case of errors, NLP processes will not be able to produce accurate results. Hence, we manually analyzed the dataset and fixed the possibly identifiable errors.

The next step would be data pre-processing, which would be done as part of the pipeline. The main aim of the pipeline is make the data ready for constituency parsing and dependency parsing. It is required that some pre-requisite annotations should be performed before doing the parsing. The steps are listed below.

1. Sentence tokenization
2. Word tokenization

3. Parts-of-speech tagging
4. Lemmatization

After performing the above pre-processing steps as part of the pipeline, the CoreDocument will be produced and available for syntactic (constituency) and semantic (dependency) parsing, followed by component extraction.

D. Component Extraction

After pre-processing is done, we can extract the components identified as part of our analysis phase. Once the components are extracted, we can assemble them in the below frame, which would be applicable for all the categories.

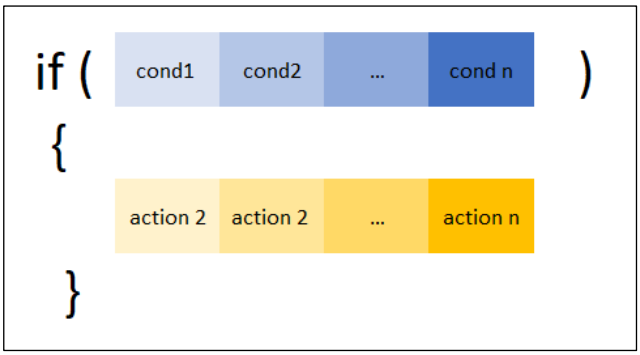

Figure 13 Frame or structure for assembling extracted components

For component extraction, we have extensively used Dependency parsing's semantic relations as they were more dependable than the outputs of constituency parsing. However, there are few places where we have used constituency parsing. Now, let us discuss how the components were extracted and the rules and patterns that were followed to extract them. Figure 14 contains a sample output snapshot that will illustrate the logic of the extraction process.

a) Actions: Actions were identified to be the main clause of a sentence. These main clauses can be specified by using the relation *nsubj* of dependency parsing. For example, let us take sentence 1 in Figure 13 and its dependency graph in Figure 16. We can see that there is only one *nsubj* relation, which helps you to identify the main clause which is nothing but the sentence's action. In requirement 34, we can observe multiple *nsubj* relations in the dependency tree in Figure 15. In this case, there can be one or more main clauses and one or more sub-ordinate or adverbial clauses.

The main clause can be differentiated from the sub-ordinate clause or the adverbial clause using the relation *advcl*. There will be an *advcl* relation between the verb of the main class and the verb of the subordinate clause, where the main clause verb is the governor. In other words, the main clause verb will never be dependent except on the root. Also, if there are multiple main clauses, the second main clause will be linked to the first main clause by a *conj* relation. However, dependency parsing arranges its relations based on the left to right logic, and hence in most cases, the second main clause occurs after the first main clause naturally.

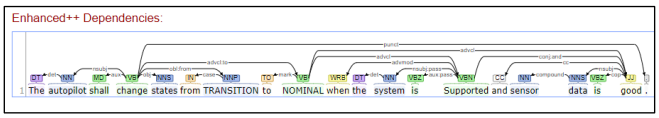

Figure 15 Dependency graph for Requirement 34

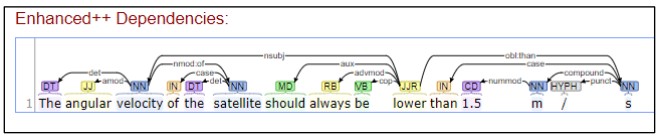

Figure 16 Dependency graph for Requirement 1

b) Subject: Subjects can be present in any clause, whether it is the main clause (action) or the sub-ordinate clause (condition). The subject can be easily identified by just extracting the dependent of the *nsubj* relation. Therefore, in requirement 1, "velocity" is the subject. But in many cases, it would be very vague to have one word as the subject, and such extracted content would not contribute much to the understandability of the requirement. We used constituency parsing to extract the immediate parent noun phrase (NP) as the subject. Therefore, from figure 17, we can say that our subject is "The angular velocity." These observed results seemed to be a lot better than the previous approach. It is still not the best since it contains unnecessary stop words and is ambiguous. It was not sure what level of noun phrase will give the required amount of information. Sometimes, moving up the tree to find a bigger NP will result in too much information, and moving down towards the leaves will result in too little information. This depth parameter is very subjective and is not easy to generalize. We then used

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
1
The angular velocity of the satellite should always be lower than 1.5 m/s
----------------------------------------------
( [ angular velocity satellite] always < 1.5  m / s  )
----------------------------------------------
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
34
The autopilot shall change states from TRANSITION to NOMINAL when the system is Supported and sensor data is good.
----------------------------------------------
IF (( [ system] when Supported  ) and ( [ sensor data] good  )  )
{
        ( [ autopilot] change [ states]  TRANSITION  -> NOMINAL  )
}
----------------------------------------------
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 14 Sample output snapshot

dependency parsing results to completely identify the subject phrase, giving much more meaningful results. After this, we can see from figure 14 final output that the subject is "[angular velocity satellite]." The methodology for extracting this will be discussed in detail in section k, 'refining subject and object phrases.'

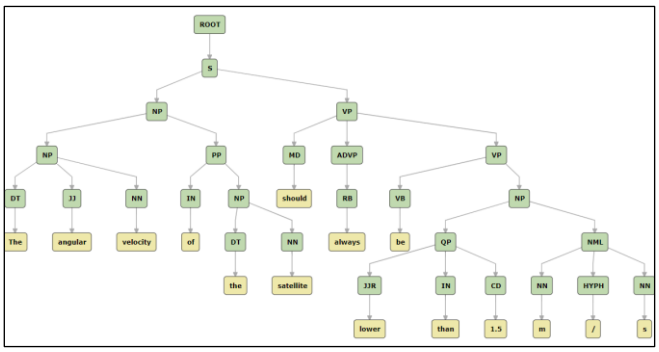

Figure 17 Constituency tree for Requirement 1

c) Root verb/comparator: The root verb or comparator can be extracted by taking the *nsubj* relation governor. This relation might sometimes occur as *nsubj:pass* when the sentence is in a passive voice. This can be an action verb, an Auxiliary verb, or a comparator verb (greater, more, less, etc.). This component is critical and can technically never be null since a sentence cannot exist without a verb. Also, all other components are extracted from relations chained from the root verb, and hence this component is the heart of our extraction process.

d) Modifier: The root verb can have one or more modifiers extracted from the dependent if the *advmod* relation that rises with the root verb as its governor. In requirement 1, the word 'always' is an adverbial modifier to the root verb 'is.' The modifiers can give more information regarding the time, manner, and how the action should occur. A negative modifier, like the word 'not,' indicates the action or condition should not happen. Therefore, modifiers are critical components.

e) Object: The object can be extracted by taking the dependant of the *obj* relation with the root verb as the governor. An object is an entity on which the do-er, i.e., the subject, performs the action. Hence, if the object information is missing, the requirement would look incomplete. Again, like the subject, a single token object might not provide much information and hence needs to be refined. This will be explained in the forthcoming sections.

f) Value: When the main clause is just a rule (as in requirement 1), the object can be replaced by a value. But this cannot be directly extracted using dependency parsing. It is quite tricky. Since value is almost always associated with numbers, we extract any CD- Cardinal in the constituency tree present inside an atomic VP. Hence, for most cases, any cardinal which is a part of the noun subject phrase will be rightly ignored and the CD that is a part of the predicate will be extracted. In requirement 1, the value 1.5 is extracted correctly by our code. However, this methodology does not guarantee accurate results for all patterns of requirements.

g) Unit: This is nothing but the metric unit of the value. For instance, in requirement 1, the unit for the value 1.5 is m/s. This occurs in pairs with value and will not happen alone. We extract the unit based on the *nummod* dependency relation with the token, which we have extracted as the value. In some cases, the unit might be a single token. But in cases like requirement 1, it consists of multiple tokens.

h) Conditions: As explained in the action section, conditions can be extracted by following the *advcl* relation, which rises from the main clause with the root verb of its governor's main clause and the subordinate clause's verb as its dependent. From the subordinate clause's verb, we can extract the subject, object, comparator, value, etc., of the subordinate clause, aka condition.

i) Conjunctions/ connectors: If there are multiple actions (main clauses) or conditions (sub-ordinate clauses), they must be grammatically separated by a conjunction. This can be identified in the dependency graph by looking for a conj or *conj:or* dependency relation. Again, the governor of this relation will be the root verb of the clause in the main clause or condition clause. The part, which comes after then colon, in the relation name will tell us what kind of conjunction it is. Since the dependency parsing result list will be arranged from left to right, we need not worry about the order if we iterate it from top to bottom. We can extract the conjunction and simply store it as part of the clause object.

j) State transition source and target value: When the main clause involves a transitional action (e.g., Requirement 34), we might have a source and target value for the state of the object or, in other words, the source and target state of the object. This can again be identified by using dependency parsing. We can lookout for the relations *obl:from, advcl:from, obl:to,* and *advcl:to* which have the root verb as their governor, and the dependent of this relation will be the respective source and target value of the state.

k) Refining the subject and object phrase: As we mentioned in the previous sections, extracting a noun component is very tricky. We would not know what level of details need to be extracted. We extracted the root subject (dependent in the *nsubj* relation) or the object (dependent in the obj relation) to improve this. We then tried to extract the dependents of the various relations like *amod, compound, nmod, nmod:if, xsub* that have the subject or object as the root. We used a simple TreeMap data structure to store the extracted token and its position in the sentence as the value-key pair. Since the TreeMap stores the data in a sorted way, we can easily print this subject phrase or object phrase in a meaningful order, which is the order in which they occurred in the actual sentence. However, even if this method might not give the best results, it is far better than the previous methods' results.

Thus, we have explained the extraction process of all the components. Once all these components are extracted, they can be simply arranged in the form shown in figure 13.

V. Testing and Evaluation

The results were evaluated manually and compared with the original natural language requirement. As per our evaluation, there were some errors due to grammatical mistakes. But, once they were corrected, around 90% of the requirements were translated with good understandability. This, however, does not include the category nine requirements, which were categorized as 'too hard or inappropriate to translate'. Since this requires more of a subjective evaluation, we could not provide reliable metrics to evaluate the work.

VI. Observations and Lessons Learned

Below are the observations and lessons learned as part of the tasks carried out as part of this project.

- Extraction of requirement extraction is a very subjective process and would require domain knowledge and expertise to evaluate and write rules.
- NLP is brittle. Almost all the existing libraries have a high expectation of the quality of the data fed to them. There is very little tolerance for errors or grammatical mistakes in the source data.
- CoreNLP is comparatively very robust, especially when it comes to parts-of-speech tagging. The tagger performed very well for almost every flavor of data.
- Semgrex and Tregex are slightly unstable. Even after following the official documentation [11] and [12], several examples threw errors despite following the same regex patterns.
- Constituency parsing was not very useful for solving our problem. We used dependency parsing to solve 90% of our extraction use cases.

VII. Threats to Validity

There are a few points that can be considered as threats to the validity of this work. We have analyzed only a fixed and limited number of requirements. Therefore, even though the extraction logic is generically written, it might not work correctly for new patterns of requirements. The next factor would be the brittle nature of NLP.

NLP libraries are written with proper grammar in mind, and hence anything unconventional would not be tolerated. The margin for error is very minimum. Also, the parsing that we used is based on the Stanford Core NLP models' performance. A dip in the quality of these models in the future might affect our results too.

VIII. Future Work

As a future work on this research, we can refine the patterns and extract more useful components. Also, we could refine the code by considering more types of requirements that can be translated. Another point would be to bring up some logic to identify and shorten phrases so that the translated logic would be less verbose.

IX. Conclusion

Therefore, we have analyzed the requirements and wrote code to translate natural language requirements into logic automatically. Requirement engineering is a significant domain, and hopefully, our project adds value to the ongoing research.

Acknowledgment

I sincerely thank my supervisor Professor Dr. Mehrdad Sabetzadeh, co-supervisor Dr. Sallam Abualhaija, and Prof. Shiva Nejati, for the support and guidance which motivated me to work on this project.

References

[1] Dresner, J., & Borchers, K. H. (1964). Maintenance, Maintainability, and System Requirements Engineering (No. 640591). SAE Technical Paper.

[2] Giannakopoulou, D., Pressburger, T., Mavridou, A., & Schumann, J. (2020, March). Generation of formal requirements from structured natural language. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 19-35). Springer, Cham.

[3] Heitmeyer, C. L., Jeffords, R. D., & Labaw, B. G. (1996). Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology (TOSEM), 5(3), 231-261.

[4] Menghi, C., Nejati, S., Gaaloul, K., & Briand, L. C. (2019, August). Generating automated and online test oracles for Simulink models with continuous and uncertain behaviors. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 27-38).

[5] Muslea, I. (1999, July). Extraction patterns for information extraction tasks: A survey. In The AAAI-99 workshop on machine learning for information extraction (Vol. 2, No. 2).

[6] Pudlitz, F., Brokhausen, F., & Vogelsang, A. (2019, September). Extraction of system states from natural language requirements. In 2019 IEEE 27th International Requirements Engineering Conference (RE) (pp. 211-222). IEEE.

[7] Allan, J., Papka, R., & Lavrenko, V. (1998, August). Online new event detection and tracking. In Proceedings of the 21st annual international ACM SIGIR conference on research and development in information retrieval (pp. 37-45).

[8] Zhou, Y., Wang, C., Yan, X., Chen, T., Panichella, S., & Gall, H. C. (2018). Automatic detection and repair recommendation of directive defects in Java API documentation. IEEE Transactions on Software Engineering.

[9] Software - The Stanford Natural Language Processing Group. Nlp.stanford.edu. (2020). Retrieved 18 December 2020, from https://nlp.stanford.edu/software/.

[10] Ramakrishnan, H. (2020). hariprasadr92/NatLangToLogic. Retrieved 22 December 2020, from https://github.com/hariprasadr92/NatLangToLogic

[11] Tregex tutorial - (2020). Retrieved 20 December 2020, from https://nlp.stanford.edu/software/tregex/The_Wonderful_World_of_Tregex.ppt/

[12] Semgrex tutorial. (2020). Retrieved 20 December 2020, from https://nlp.stanford.edu/software/Semgrex.ppt