Formalization of Natural Language Text using Syntactic Dependency

Keleş, Işilsu

Award date:
2021

Technische Universiteit Eindhoven University of Technology

Department of Mathematics and Computer Science
Process Analytics Group

Formalization of Natural Language Text using Syntactic Dependency

Master Thesis

Data Science Program

Işılsu Keleş

Supervisors:
Dr. Natalia Sidorova

Assessment Committee Members:
Dr. Natalia Sidorova
Fatemeh Shafiee, M.Sc.
Bas Luttik, dr.

Eindhoven, October 2021

Abstract

For millennia, humans have used natural language as a medium to convey information to one another. The complexity of the information has been ever increasing, and so has the complexity of the language conveying it. The rise of the digital age has exacerbated that effect as we witness the explosion of data, especially in textual format. Most of that text, however, remains unstructured and hard to interpret by computers. This thesis aims to bring natural language text closer to the machine's understanding through formalization with the purpose of enabling many useful applications such as formal reasoning and model checking [51].

We are interested in a particular type of texts—those which describe a procedure such as instruction manuals. *Procedural descriptions* usually contain an embedded structure which can be categorized as a business process. A *business process* is a series of ordered steps and decisions that describe the way a procedure is complete. For example, issuing legal documents, registering a patient, taking a regular medication, and so on. Therefore, such embedded structure can be captured through process modeling languages such as BPMN (Business Process Model and Notation), or it can be captured through temporal logics such as LTL (Linear Temporal Logic).

This project aims to lay the foundation for a framework that can extract fragments of process information from procedural description texts and translate them into a formal language such as the aforementioned LTL. This project also aims to solve the challenge of detecting multiple linguistic constructs within one sentence even when some are nested within others.

Our proposed design consists of five modules. Module 1 prepares the text by segmenting sentences and performing preliminary linguistic processing. Module 2 performs syntactic dependency analysis to detect the syntactic relations between words in a sentence. The output is a dependency tree carrying the relations. Implicitly, the dependency relations inform us of which linguistic construct is nested within which. Module 3 detects the linguistic constructs that are known to carry process information. Module 4 transforms the detected constructs into a formal intermediate representation we call a linguistic-semantics (LingSem) tree. Module 5 translates the LingSem tree into a formal expression.

To validate the results of this approach, the intermediate and final formal outputs of the implemented design are compared to the formalization produced by a human participant. We found that our intermediate representation output is accurate, but we also found many discrepancies between human and machine in the final LTL output.

Preface

I would like to thank Natalia Sidorova for her supervision and guidance throughout this thesis. Her advice helped me steer this work into a suitable direction and grasp the context and possible solutions. I would like to thank Fatemeh Shafiee for providing both academic and emotional support. She has been like a sister to me during this time. Additionally, I would like to express my gratitude to my colleagues and friends, whose encouragement and feedback kept me motivated. Finally, I would like to express my gratitude to my parents, who have cared for me throughout my entire academic career.

Contents

Contents v

List of Figures viii

List of Tables x

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Question . . . 2
- 1.3 Research Approach . . . 2
- 1.4 Outline . . . 3

2 Preliminaries 5
- 2.1 Natural Language . . . 5
  - 2.1.1 Natural Language Processing . . . 5
  - 2.1.2 Dependency Grammars and Dependency Parsing . . . 8
- 2.2 Business Processes . . . 9
- 2.3 Formal Languages . . . 9
  - 2.3.1 Linear Temporal Logic . . . 9
  - 2.3.2 Traces and Trace Notation . . . 10
  - 2.3.3 Process Templates and DECLARE . . . 11
  - 2.3.4 Informal Semantics . . . 12

3 Background and Related Works 13
- 3.1 Natural Language to Declarative Process Models . . . 13
- 3.2 Natural Language to Business Process Models . . . 14
- 3.3 Natural Language to Linear Temporal Logic . . . 16
- 3.4 Process Extraction from Text . . . 16
- 3.5 Annotation of Procedural Texts . . . 17
- 3.6 Formalization of Software Requirements . . . 18
  - 3.6.1 Software Design Documents to Requirements . . . 18
  - 3.6.2 Informal Specifications . . . 18
- 3.7 Tools . . . 19

4 Solution Approach **21**
4.1 Module 1: Preprocessor . . . 22
4.2 Module 2: Syntactic Dependency Parser . . . 23
4.3 Module 3: Dependency Pattern Matcher . . . 26
4.3.1 Dependency Patterns . . . 26
4.3.2 The Procedure Followed for Crafting a Dependency Pattern . . . 29
4.3.3 Output of Module 3 . . . 34
4.4 Module 4: Linguistic Parse Tree Builder . . . 34
4.4.1 Task 1: Predicate Construction . . . 35
4.4.2 Task 2: Building a LingSem Tree . . . 39
4.5 Module 5: Translator . . . 45
4.5.1 Determining the Desired Output Formalism . . . 46
4.5.2 The LTL Dictionary . . . 46
4.6 Resolution to Research Question . . . 47
4.7 Limitations and Threats to Validity . . . 48
4.7.1 Limiting our Observation to a Specific Domain . . . 48
4.7.2 One Dependency Pattern Mapping to Multiple Meanings . . . 48
4.7.3 Relying Solely on Dependency Analysis . . . 49
4.7.4 The Limitation of Relying on Function Words Alone . . . 49
4.7.5 Relying on Human Interpretation . . . 49
4.8 Further Discussions . . . 49
4.8.1 The Accuracy of the Dependency Parser (Module 2) . . . 50

5 Results and Evaluation **53**
5.1 Case Study Text . . . 53
5.2 Result Analysis Procedure . . . 55
5.3 Results . . . 55
5.3.1 Sentence $\mathcal{S}_1$: . . . 55
5.3.2 Sentence $\mathcal{S}_2$: . . . 56
5.3.3 Sentence $\mathcal{S}_{13}$: . . . 59
5.4 Discussion of Results . . . 60

6 Conclusion **63**
6.1 Contributions . . . 63
6.2 Results . . . 63
6.3 Future Work . . . 64
6.3.1 Using Co-reference Resolution to Relate Multiple Sentences . . . 64
6.3.2 Defining More Transformation Rules . . . 64
6.3.3 Probabilistic Precedence Rules . . . 65
6.3.4 Quality Control Measures . . . 65
6.3.5 LingSem Definition . . . 65
6.4 Concluding Remarks . . . 65

Bibliography **67**

Appendix **73**

A Results of Secondary Significance **73**
A.1 Sentence $\mathcal{S}_3$: 73
A.2 Sentence $\mathcal{S}_4$: 74
A.3 Sentence $\mathcal{S}_5$: 75
A.4 Sentence $\mathcal{S}_6$: 76
A.5 Sentence $\mathcal{S}_7$: 77
A.6 Sentence $\mathcal{S}_8$: 77
A.7 Sentence $\mathcal{S}_9$: 78
A.8 Sentence $\mathcal{S}_{10}$: 79
A.9 Sentence $\mathcal{S}_{11}$: 79
A.10 Sentence $\mathcal{S}_{12}$: 80
A.11 Sentence $\mathcal{S}_{14}$: 80
A.12 Sentence $\mathcal{S}_{15}$: 81
A.13 Sentence $\mathcal{S}_{16}$: 82
A.14 Sentence $\mathcal{S}_{17}$: 82
A.15 Sentence $\mathcal{S}_{18}$: 83

B Dependency Trees of Ibuprofen Usage Text **85**

C Output of Python Tool **92**

List of Figures

2.1 An example sentence illustrating key concepts of sentence structure in natural language: "Emma may cross the wide street only when it is empty". 6
2.2 A dependency-style parse (left) next to the corresponding constituent-based analysis (right) for the sentence "I prefer the morning flight through Denver" (source: [26]). 8
2.3 A mapping from some of the process templates used by DECLARE to LTL formulas (Source: [41] Fig 6.5). 12

3.1 High-level methodology of Shafiee [53]. Image taken from the same paper 14
3.2 High-level methodology of Honkisz et al. [25]. Image taken from the same source. . 15
3.3 Comparison of functionalities offered by SPACY, NLTK and CoreNLP. (Source: [64]) 20

4.1 Flowchart showing the high-level design of the tool with the contribution of this thesis highlighted. 21
4.2 Dependency tree of $\mathcal{S}_2$; unmodified. 25
4.3 Dependency tree of $\mathcal{S}_{2_1}$ which is $\mathcal{S}_2$ simplified by replacing the *before* adverbial clause by the adverb *now*. 25
4.4 Dependency tree of $\mathcal{S}_{2_2}$, which is $\mathcal{S}_2$ modified by adding the disambiguating complementizer *that*. 25
4.5 The dependency pattern specified to detect the *and* conjunction of two verbs. . . . 27
4.6 Dependency tree of "Help yourself before others". 33
4.7 Dependency tree of "Stand still before your physician". 33
4.8 Dependency tree of "Take one dose before dinner". 33
4.9 Tokens from the dependency tree of Figure 4.2 that are matched against dependency patterns. 38
4.10 Tokens from the dependency tree of Figure 4.2 that are matched against dependency patterns, with the three matches highlighted. 39
4.11 Dependency tree of the sentence "If you read and learn, you become smarter". . . . 41
4.12 Dependency tree from Figure 4.11 with pattern matches marked. 41
4.13 The LingSem tree constructed from the dependency tree and matches shown in Figure 4.12. 41
4.14 Dependency tree for sentence $\mathcal{S}_s$: "If you are smart, listen and think, before you talk" 42
4.15 The LingSem tree of Sentence $\mathcal{S}_2$ before introducing the precedence rules. 44
4.16 The LingSem tree of Sentence $\mathcal{S}_2$ after introducing the precedence rules. 44

5.1 An alternative LingSem tree for $\mathcal{S}_2$. 59

A.1 Dependency tree of $\mathcal{S}_5$. 76
A.2 Dependency tree of $\mathcal{S}_5$ after switching around the main clause with the *after*-clause. 76

B.1 Dependency tree of $\mathcal{S}_1$. 86
B.2 Dependency tree of $\mathcal{S}_2$. 86
B.3 Dependency tree of $\mathcal{S}_3$. 86
B.4 Dependency tree of $\mathcal{S}_4$. 87
B.5 Dependency tree of $\mathcal{S}_5$. 87
B.6 Dependency tree of $\mathcal{S}_6$. 87
B.7 Dependency tree of $\mathcal{S}_7$. 88
B.8 Dependency tree of $\mathcal{S}_8$. 88
B.9 Dependency tree of $\mathcal{S}_9$. 88
B.10 Dependency tree of $\mathcal{S}_{10}$. 89
B.11 Dependency tree of $\mathcal{S}_{11}$. 89
B.12 Dependency tree of $\mathcal{S}_{12}$. 89
B.13 Dependency tree of $\mathcal{S}_{13}$. 90
B.14 Dependency tree of $\mathcal{S}_{14}$. 90
B.15 Dependency tree of $\mathcal{S}_{15}$. 90
B.16 Dependency tree of $\mathcal{S}_{16}$. 91
B.17 Dependency tree of $\mathcal{S}_{17}$. 91
B.18 Dependency tree of $\mathcal{S}_{18}$. 91

List of Tables

4.1 Output of Module 1 for the sample sentence $\mathcal{S}_2$. 23
4.2 The dependency-relation labels shown on various dependency trees throughout the thesis and the full-form term of each. 24
4.3 The dependency pattern matching rules defined in our framework. 29
4.4 Some of the semantics of interest for Module 3, expressed in process templates, one linguistic equivalent for each and the equivalent LTL formula. 30
4.5 Matched tokens from Figure 4.10, their predicates and the subtractions performed on each to achieve the predicate. 38

5.1 All sentences comprising the sample input text of Ibuprofen usage. 54

Chapter 1

Introduction

For millennia, humans have used natural language as a medium to convey information to one another. The complexity of the information has been ever increasing, and so has the complexity of the language conveying it. That is especially observable in the most recent decades, with the rise of the digital age, and the explosion of information. With so much data storage in today's world, much of the complex information is stored as natural language text. Most of that text, however, remains unstructured and hard to interpret by computers. Formalization of natural language brings it much closer to a computer's understanding [51]. Computer understanding of natural language enables many useful applications such as formal reasoning, knowledge extraction, indexing and searching, discourse analysis, and sentiment analysis [51].

This thesis is concerned with texts describing procedural instructions such as medicine leaflets, instruction manuals, cooking recipes, or business contracts. A simple yet extensible framework is developed to offer automated formalization of text through the use of state-of-the-art tools and standards alongside user-defined rules and dictionaries. In this chapter, we introduce the basic background and provide an overview of the whole project. In Section 1.1, we explain why formalization of text is desirable. Section 1.2 presents the research question of this thesis. Section 1.3 introduces our approach. The further outline of this thesis is described in Section 1.4.

1.1 Motivation

Formalization of natural language serves many useful purposes. Consider texts describing software requirements. Translating requirements to a formalism such as finite state machines, for instance, enables formal verification of properties of the software design.

Next to software requirements, other types of texts fall within the interest of this thesis, i.e., instruction guides, user manuals, software documentation, informal semantics, or informal specifications of a system. We generally label any such text as a *procedural description*. In addition to the aforementioned benefits of formalization, translating a procedural description to, say, the well-known Linear Temporal Logic (LTL), enables formal reasoning and querying using logic formulas. For example, we can answer inquiries about the sequence of activities within a procedure,

about how two activities are related or about which activities carry certain attributes. Furthermore, instructions can be re-generated with more detail or more context provided [68].

The extent of inquiries one can make with regards to a procedural description is vast since there are many semantic elements embedded within a procedural text. Several works have focused on semantic elements captured by business processes. A *business process*, or a *process* for short, is a series of ordered steps and decisions that describe the way a procedure is complete [70].

Needless to say, manually processing the humongous amount of existing documents is beyond our capacity as humans, not to mention slow and error-prone. This reality urges the need for automating such a process using intelligent computing.

The next section explains the objective of this thesis in light of that.

1.2 Research Question

The aim of this project to extract process information from textual data. The purpose is to enable the machine to perform advanced computations such as process mining on such data. Therefore, the goal can be formulated as follows:

> **Goal: To extract process information from textual data.**

There are many challenges in processing textual data because a single process is usually described in multiple sentences with semantic relations between them. Therefore, to limit our scope, we decided to focus on process information that lies within an individual sentence.

Moreover, even a single sentence can be very complex especially when it contains nesting, i.e., when a clause has another clause embedded inside it. For example, in the sentence "If you study before taking the exam, you pass", the *before*-clause is nested within the *if*-clause. We decided to focus on detecting such nesting within a sentence with no particular limit to its depth or the amount of clauses in a sentence.

If we think in terms of process templates, the templates expressing the linguistic constructs of *if* and *before* in the aforementioned sentence are called the *Response* and the *Precedence* templates respectively. There are various process templates and various sets of defined templates. For time constraints, we decided to limit the number of templates to focus on.

Lastly, while a single sentence may not be enough to contain an entire process, it can contain process fragments, each of which can be expressed through one of the defined process templates.

To conclude, these aspects lead to the following **research question**:

> **How can nested process fragments be extracted from a sentence describing a procedure in natural language?**

1.3 Research Approach

The main goal of this project is to extract process model information from natural language sentences, including ones that contain nesting.

To achieve this goal, the following steps were taken.

Literature review. At the beginning and during the project, we did a thorough examination of the related work concerning process extraction from text and reviewed the solutions related to our problem. We explored the transformation of natural language into different process models and LTL. This research helped us clarify the vision and learn more about the state of the art on process extraction and on the formalization of natural language.

Case study. We use a dataset collected by Shafiee [53]. The data was a corpus of medicinal usage texts gathered from WebMD[1]. One particular piece of text we focused on is found at WebMD: Ibuprofen Use[2].

Tools. To study dependency trees, we benefited greatly from spaCy[3], an open-source NLP framework for the Python programming language. It includes built-in pipelines, trained models, and extensions. These perform tasks as text processing, sentence segmentation, dependency parsing and visualizing the output beautifully. We decided to use spaCy for its accuracy of built-in models, ease of use and excellent documentation. Moreover, we benefited from other Python libraries such as NLTK[4] for more NLP operations.

Method Based on the literature review, we decided to use dependency parsing as our primary technique for language analysis. Observing the resulting dependency trees, we noted specific repeating patterns that are associated with certain semantics of natural language, e.g., the function word *if* connecting two clauses entails a conditional statement. We defined a set of these repeating patterns, called dependency patterns, and defined a mapping from each pattern to a formal expression which was derived essentially from human understanding. Next, we started searching for the patterns in other sentences. Matches within a sentence are arranged in a nested hierarchy based on the nesting found in the dependency tree. Finally, based on the aforementioned mapping, the matches are transformed into a final formal expression.

Evaluation of the results. The last phase of this project is to analyze and evaluate the results. This step brought much insight to our work. It revealed many flaws in the initial design stages and assumptions. Furthermore, it greatly improved our understanding of dependency parsing and brought many ideas for improvement and future work. We carried out the evaluation by comparing the LTL output of our implementation with that of a human participant.

1.4 Outline

Chapter 2 provides relevant concepts and definitions necessary for understanding the rest of the project. Chapter 3 provides the background, describes the works and literature related to our problem and examines common challenges and techniques. Chapter 4 demonstrates our approach

[1] https://www.webmd.com/
[2] https://www.webmd.com/drugs/2/drug-5166-9368/ibuprofen-oral/ibuprofen-oral/details
[3] https://spacy.io
[4] https://www.nltk.org

to extracting process fragments from text and explains the design and implementation of the framework in five modules. Chapter 5 shows the results of the solution and discusses and evaluates them. Finally, Chapter 6 summarizes the thesis and presents conclusions and possible improvements for future work.

Chapter 2

Preliminaries

In this chapter, we introduce the relevant concepts and definitions necessary for understanding the coming chapters.

2.1 Natural Language

The term natural language refers to human languages in general that naturally evolved away from deliberate artificial design. Human evolved languages are, thus, *natural* as opposed to programming languages or human-constructed languages such as Esperanto, Lojban and even Klingon [40].

Natural language is filled with ambiguities that make it incredibly difficult for computers to understand. In many instances, the intended meaning of an utterance is implicit and hidden behind layers of non-linguistic knowledge. Homonyms, homophones, sarcasm, idioms, metaphors, grammar and usage exceptions, variations in sentence structure——these are but a few of the irregularities of natural language that take humans years to learn.

For the purpose of this thesis, we only address natural language in its textual form.

2.1.1 Natural Language Processing

As the application of Artificial Intelligence expands into our daily lives, the processing of natural language by a machine is an evermore increasing necessity—one that has given rise to the field of Natural Language Processing (NLP), also known as Natural Language Understanding (NLU). The term NLP refers to a branch of Artificial Intelligence concerned with granting computers the ability to understand/interpret speech and text in a way that can parallel human understanding. The following explains concepts of natural language that are used throughout this thesis [36, 26], followed by examples. Additionally, Figure 2.1 illustrates an example sentence marking these concepts.

Linguistic constructs: These are parts of linguistic tokens that adhere to a specific syntax in accordance with the rules of a language. Statements such as conditionals (e.g., *if else*), *for* (*each*)

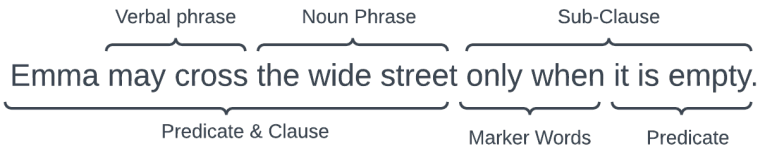

Figure 2.1: An example sentence illustrating key concepts of sentence structure in natural language: "Emma may cross the wide street only when it is empty".

loops, and *while* loops are part of the linguistic construct of a programming language. These constructs are essential to ensure that instructions are clearly interpreted by the machine.

Phrases: These are a group of words, consisting of one word or a complete sentence, that form a grammatical unit and plays a particular role within the syntactic structure of a sentence. One can construct so-called phrase 'trees', where each sub-tree can be considered as a phrase.

Predicates: These are either all the words in a standard declarative sentence or simply the main content verb or associated expression of a clause. Both definitions may be used.

Clauses: These are a group of at least one word functioning as a single stand-alone unit, typically consisting of a semantic predicate. The most common clause in the English language is the subject-verb-clause, e.g., "I go to work every day". Commonly in English, a sentence may consists of a pair of clauses, one being a *main clause* and the other a *subordinate clause* or *sub-clause*. The main clause can stand alone as a complete meaningful sentence, but a sub-clause depends on the main clause to be meaningful. As shown in Figure 2.1, the clause headed by marker words is the sub-clause.

Marker Words: These are function words such as *if*, *before*, *when*, *until*, *while*, etc., that usually come at the head of a clause marking the type and function of the clause.

Examples about Predicates, Clauses, and Marker Words: Since we use these terms in the rest of this thesis, comprehending them through a few examples should prove fruitful. Take the sentence "Emma crosses the street". This entire sentence is a single predicate. It is also a clause, but it could also have a dependent sub-clause as in "Emma crosses the street, only when there is no traffic". Here, there are two predicates: "Emma crosses the street" and "there is no traffic". The function words "only when" are considered the marker of the sub-clause. Another possibility is that the Emma predicate could itself be a dependent sub-clause as in "Once Emma crosses the street, we start moving".

Nested hierarchy These are tree-like structures that show direct or indirect relations between entities in a set. In the context of NLP, nested hierarchies can show direct or indirect relations or dependencies between a word and another word or phrase.

Levels of NLP

There are different levels of analysis of language [18]. In order from least to most complex, these are:

1. **Morphological Analysis**, which deals with the study of word structures and formation, and focuses on analysing individual components of words. For example, "unhappiness" can be broken down into three units (prefix *un-*, refers to not being; stem *happy*; suffix *-ness*, refers to a state of being).

2. **Lexical Analysis**, which deals with studying words with respect to their part-of-speech and meaning. For example, the word *duck* can be both a verb or a noun, but its part-of-speech and meaning depends on its context with other words in a sentence.

3. **Syntactic Analysis**, which refers to grouping words into phrase and clause brackets based on the part-of-speech tagging output from the lexical analysis. This analysis (also known as parsing) allows to extract phrases that convey more meaning compared to each word individually.

4. **Semantic Analysis**, which deals with determining the meaning of a sentence by relating syntactic features and removing ambiguity in words based on the context.

5. **Discourse Analysis**, which deals with analysing the structure of text on a multi-sentence level by attempting to make connections between words and sentences.

6. **Pragmatic Analysis**, which deals with using real-world knowledge (such as time and location mentioned in text) and understanding how this knowledge may impact the meaning of what is being communication.

Techniques of NLP

There are a handful of techniques that are used in NLP that operate on one or more of the aforementioned levels [15]:

1. **Part-of-Speech (POS) tagging**, which is the process of marking words in a text corresponding to its definition and context in the sentence. A prime example of POS tagging is identification of grammatical cases (nouns, pronouns, verbs, etc.). This technique mainly performs a lexical analysis of a sentence or text.

2. **Dependency Analysis**, which is the process of extracting dependencies between entities in a sentence, forming directional relationship between these entities. This technique mainly performs a syntactical analysis of a sentence.

3. **Named Entity Recognition**, which is the process of locating and classifying entities into pre-defined categories such as names, locations, time expressions, and so on. This technique mainly works on a pragmatic level.

4. **Co-reference Resolution**, which refers to the process of finding all expressions referring to the same entity in a text. This technique mainly operates on the discourse level of analysis.

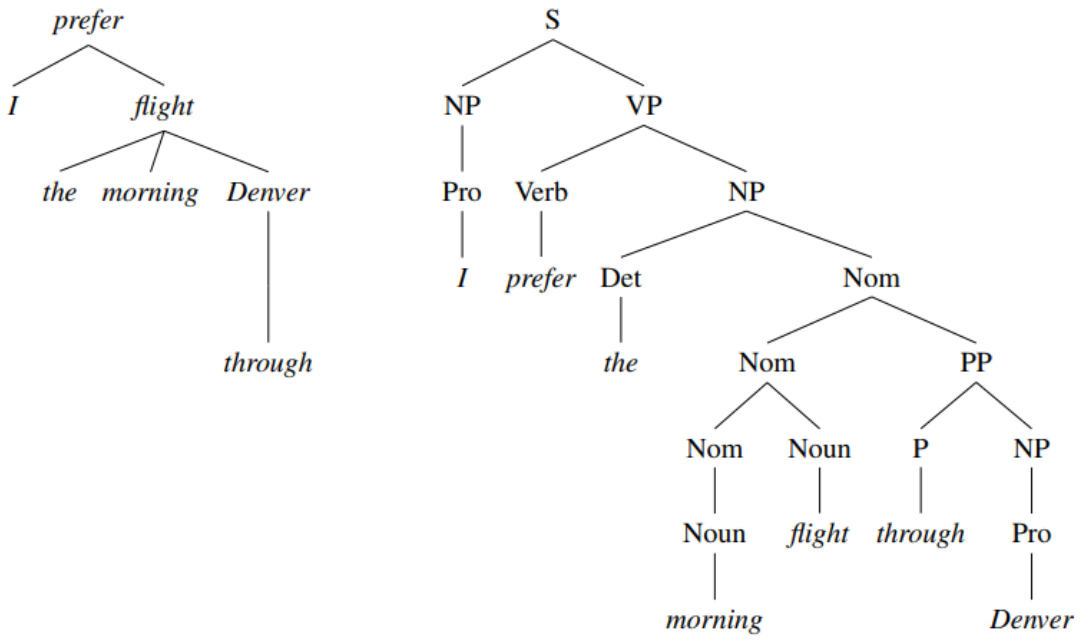

Figure 2.2: A dependency-style parse (left) next to the corresponding constituent-based analysis (right) for the sentence "I prefer the morning flight through Denver" (source: [26]).

2.1.2 Dependency Grammars and Dependency Parsing

In modern grammatical theories, dependency grammars assume that all linguistic units, e.g. words, are connected to each other by a dependency relation via directed links. Sentences can be processed according to dependency grammar standards via dependency parsing [39]. The dependency roots from the verb of the clause structure, and each other syntactical unit is either directly or indirectly connected to the verb, forming dependencies. Dependency parsing is different from constituency parsing (or syntactical parsing), as the latter displays the syntactic structure of a sentence using context-free grammar [26]. Thus, a major advantage of dependency parsing over syntactical parsing include the ability to deal with morphologically-rich languages with a relatively free word order [26]. An example of how a sentence can be parsed using both parsing strategies is depicted in Figure 2.2.

It is important to note that dependency parsing is capable of capturing nested clauses/phrases down to an arbitrary depth [26].

There are two types of dependency parsing, namely transition-based and graph-based dependency parsing. Transition-based parsing defines a transition system for mapping a sentence to its dependency graph. Given an induced model for predicting the next state transition, the parser constructs an optimal transition system using a greedy approach [26]. Graph-based parsing defines a space of candidate dependency graphs for a given sentence, induces a model to score each dependency graph, and takes the highest-scoring model as the final dependency graph. This type of parsing is based on an exhaustive approach and is typically more accurate [26]. For that reason, we decide to use the latter type, graph-based parsing.

Definition of a Dependency Tree. A *dependency tree* is essentially a tree (the well-known hierarchical structure [1]) of nodes and edges used to show the syntactic dependence of words in a sentence on one another. In a dependency tree, nodes represent tokens (individual words or phrases) while edges represent dependency relations. An edge directed from parent x to child y carrying a label l (notation: $x \xrightarrow{l} y$) means that token y depends on token x with a dependency relation labelled l such that the label l belongs in a finite set of dependency relations.

An example of a dependency tree is given through a sample output of Module 2 in Section 4.2.

2.2 Business Processes

A business process refers to a collection of related and structured tasks or activities which produces a service or product [70]. In the context of this work, a process usually refers to a business process unless stated otherwise. The activities in a process are carried out by employees or equipment and are usually atomic, meaning that the activity cannot be split up into sub-activities. Activities can be accompanied by an event, which is a specific instance of time that marks the start or end of an activity [70].

Process discovery is one of the key steps in process mining [71] and focuses on discovering a set of activity sequences indicating the start of a process to its termination [53]. In text, a process can span multiple sentences and one sentences may not hold an entire process from start to end, but contain a fragment of a process.

In this project, we are interested in discovering fragments of processes in sentences.

2.3 Formal Languages

A formal language is a well-defined language with a well-formed alphabet that follows a specific set of syntactic and semantic rules [52]. Interestingly enough, the theory of formal language was born from the field of linguistics for the purpose of defining a universal language [17]. Formal language expressions are a valuable means for many advanced computing tasks such as model checking and formal reasoning [35].

In the following, we explain two formal languages we use in our approach, DECLARE and LTL.

2.3.1 Linear Temporal Logic

Linear-time temporal logic (LTL) is a logical formalism which is commonly adopted in formal verification, in particular in consistency and model checking, and it can be a very useful tool for capturing the temporal content of natural language utterances. Such capabilities can be exploited in many important applications, including disambiguation of technical requirements, log analysis, and automated reasoning over temporal data [6, 43].

LTL formulas are composed of the following elements [47]:

- a finite set of atomic propositions *Prop*.

- the Boolean connectives $\neg, \wedge, \vee, \rightarrow$.
- the following temporal connectives:
  - $\mathcal{U}$ (until).
  - $\mathcal{R}$ (release).
  - $\bigcirc$, also called $\mathcal{X}$ for 'neXt time-step'.
  - $\Box$, also called $\mathcal{G}$ for 'Globally'.
  - $\Diamond$, also called $\mathcal{F}$ for 'in the Future', also called 'eventually' and 'some time'.

Rozier [47] explains the meaning of these operators intuitively as follows, given two propositions ψ and ϕ:

- $\psi \; \mathcal{U} \; \phi$: Either ϕ is true now or ψ is true now and ψ remains true *until* such a time when ϕ holds.
- $\psi \; \mathcal{R} \; \phi$: Said as ψ releases ϕ. This signifies that ϕ must be true now and remain true until such a time when ψ is true, thus *releasing* ϕ.
- $\mathcal{X} \; \psi$: This means ψ is true in the *next* time step, i.e. immediately after the current one.
- $\Diamond \; \psi$: This means ψ must either be true now or at some future time step. Thus, it is *eventually* true.
- $\Box \; \psi$: This means ψ is true in every time step.

There are various extensions of LTL. We are interested in an extension of two operators: $\mathcal{W}$ (Weak until) and $\mathcal{M}$ (Strong release). Following is the intuitive explanation [74, 30]:

- $\psi \; \mathcal{W} \; \phi$: This means that ψ has to hold at least until ϕ is true. That is, if ϕ never becomes true, ψ must remain true forever.
- $\psi \; \mathcal{M} \; \phi$: This means that ϕ has to be true until and including the point where ψ first becomes true, which must hold at the current time or at a future time.

2.3.2 Traces and Trace Notation

In propositional logic, the semantic interpretation of a propositional expression is given by a boolean valuation of the expression, i.e., *true* or *false*. In LTL, this concept is extended such that an LTL formula is evaluated over a sequence of states. We assume the reader's familiarity with state transition systems and with LTL. For the complete formal context, we refer the reader to Kröger and Merz [30] (Sections 2.1 and 6.2 of the book). For the purpose of this thesis, it suffices to say that for any LTL formula, there is an underlying temporal structure. A *trace*, called a run in [30], is a sequence of states in the given temporal structure. Each state is defined by a propositional valuation; a *propositional valuation* is described by the set of variables it makes true. For example, given that A, B and C are variables representing atomic propositions, the set $\{A, B, \neg C\}$ is a single propositional valuation and it represents a state. In our notation, a trace is

shown as a sequence of propositional valuations delimited by the $\rightarrowtail$ operator representing a state transition. Examples are given below.

Given an LTL formula f and a trace t, we say that t satisfies f and that f accepts t if and only if the semantic interpretation of f is *true* when f is evaluated over t.

Assuming there is a finite number of traces that satisfy f, we say that $T = Traces(f)$ is the set of all traces that satisfy f.

Assuming that a sentence $\mathcal{S}$ can be expressed as a finite set of traces T', we say that an LTL formula f is an acceptable translation of $\mathcal{S}$ if $T' = Traces(f)$. To show that f is not an acceptable translation of $\mathcal{S}$, it suffices to provide a counterexample trace t' that satisfies f but is not expressed by S or to provide a counterexample trace t'' that is expressed by $\mathcal{S}$ but not accepted by f. Such counterexample traces aid the discussion in Chapter 5.

When discussing the LTL translation of sentences, such as in Chapter 5, we use all of the concepts mentioned above in addition to the trace notation. For example, consider the LTL formula $f = \mathcal{F}(A \wedge \neg B)$. Then, the trace $t_1 = \{A, B\} \rightarrowtail \{A, B\} \rightarrowtail \{A, B\}$ does not satisfy f because it terminates without ever reaching a state that satisfies $A \wedge \neg B$. We also say that f does not accept t_1. On the other hand, the trace $t_2 = \{A, B\} \rightarrowtail \{A, B\} \rightarrowtail \{A, \neg B\}$ does satisfy f. Notice that the value of A in the first two states of t_2 does not affect the satisfiability of t_2. In such cases, for the purpose of brevity, our notation allows dropping irrelevant literals, the literal A in this case, effectively assigning a *don't-care* value. Then, we write the trace as $t'_2 = \{B\} \rightarrowtail \{B\} \rightarrowtail \{A, \neg B\}$.

For simplicity, we do not consider infinite traces.

2.3.3 Process Templates and DECLARE

Dwyer et al. [12, 13] proposed a system of patterns that specify properties to be used for the purpose of formal specification. Throughout the thesis, we refer to these patterns as *process templates*. They defined several process templates that fall within three categories [53]:

1. Occurrence templates such as Existence, Absence, Bounded Existence, and Universality, concern the occurrence of an event or state in the system.

2. Order templates express the relative order in which events or states occur. They include Precedence and Response.

3. Chain templates express complex combinations relationships between events or states. These include Chain Precedence and Chain Response.

Dwyer et al. provide a mapping from their process templates to LTL given in [14].

These templates were later adopted into a a declarative language, called DECLARE, proposed by Pesic and Van der Aalst in [41]. In DECLARE, classes of properties and constraints are represented abstractly by the aforementioned process templates. A DECLARE model consists of a set of constraints based on those templates. Pesic and Van der Aalst also provide a mapping to LTL, based on Dwyer's mapping, shown in Figure 2.3.

We explain the meanings of some of these templates. $Response(x, y)$ dictates that if x occurs, then y must eventually occur. $Precedence(x, y)$ dictates that y cannot occur if x has not occurred earlier. $Coexistence(x, y)$ dictates that if either x or y occurs at least once, then the other must eventually occur at least once.

$$
\begin{aligned}
&\mathit{existence}(A) && \equiv \Diamond A \\
&\mathit{init}(A) && \equiv A \\
&\mathit{response}(A, B) && \equiv \Box(A \Rightarrow \Diamond B) \\
&\mathit{precedence}(A, B) && \equiv (\neg B) W A \\
&\mathit{succession}(A, B) && \equiv \mathit{response}(A, B) \ \wedge \ \mathit{precedence}(A, B) \\
&\mathit{not\ co\text{-}existence}(A, B) && \equiv \neg(\Diamond A \wedge \Diamond B)
\end{aligned}
$$

Figure 2.3: A mapping from some of the process templates used by DECLARE to LTL formulas (Source: [41] Fig 6.5).

2.3.4 Informal Semantics

The term Informal Semantics can only be defined in the context of formal semantics. The *informal semantics* of a formal system or a formal language can be defined as a means to express the knowledge of the system that is conveyed by its theories and formulas in a precise and systematic way [11]. Examples of informal semantics would be our explanation of the function of certain LTL operators and process templates in Sections 2.3.1 and 2.3.3 respectively.

Chapter 3

Background and Related Works

In this chapter, we provide the background knowledge and related work concerning formalization of natural language and review the solutions related to our problem. Moreover, we explore the transformation of natural language to different process models and LTL.

3.1 Natural Language to Declarative Process Models

Several approaches have been developed for constructing declarative process models from natural language. Van der Aa et al. [69] recently proposed an automated approach for extracting declarative process models using natural language. Their approach involves linguistic processing for identifying semantic components, activity extraction for capturing sequential actions, and constraint generation, which can handle descriptions with multiple constraints. Furthermore, they map naïve grammatical constraints to DECLARE statement templates. Studying these constraints is useful because it acts as a reference when wanting to cover as many syntactic patterns as they did. They also created constraint classes in order to group similar constraints. For example, an activity can be found in text as a verb: "create a ticket", or as a noun: "creation of a ticket". Thus, the two constraints matching these two phrases belong in the activity class.

Riefer et al. [46] provided an overview of state-of-the-art approaches for text-to-model mining. These approaches were compared against each other based on the following factors: their textual inputs accepted (strict or flexible), NLP techniques used (syntactic analysis and/or semantic analysis), and models generated (e.g., using activities, events, and/or actors). These comparisons are useful to study in order to, potentially, address their shortcomings in this work. Based on the comparisons, the following pros and cons for each state-of-the-art approach were discovered:

- Process Discovery from Model and Text Artefacts [22]: The main advantage of this approach is that it makes use of reference models, i.e., an interlinked set of components that are clearly defined; however, this approach can only create parts of a model, and connections cannot be made if no coherence is found.
- Business Process Mining from Group Stories [23]: This approach is flexible and works for

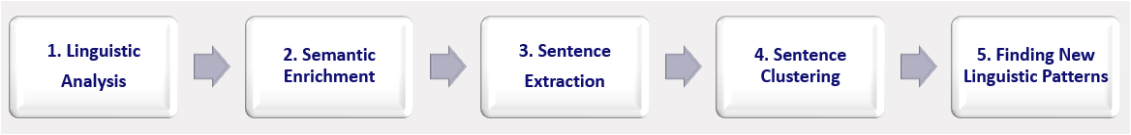

Figure 3.1: High-level methodology of Shafiee [53]. Image taken from the same paper

text-to-speech input; however, there is no semantic analysis involved and the generated BPMN models are not complete.

- Use Cases to Process Specifications in Business Process Modeling Notation [54]: This approach is very flexible about speech and domain, and can compute models relatively fast compared to other approaches; however, there is no clarity about the input text format, and the approach requires effort and domain knowledge to operate properly.
- Process Model Generation from Natural Language Text [20]: This approach creates complete models and has a thorough semantic analysis. No disadvantages could be concluded from the study.
- Automatic Process Model Discovery from Textual Methodologies [16]: This approach does not require a list of keywords unlike other models; however, this approach does not create any model whatsoever.

The work of Shafiee [53] represents the starting point for this thesis. Shafiee developed a framework to mine Local Process Models (LPM) from Natural Language. She gathered data from WebMD describing the usage of medicines. Her work can be summarized in steps which were nicely visualized by the author in Figure 3.1. First, she uses NLP tools to analyze the data and annotate it with POS tags before she annotates tokens with semantic tags using semantic ontology resources. After that, she applies rules specified in regular expressions to detect certain linguistic patterns from each sentence. Next, she applies a semantic similarity measure on all sentences in order to cluster them. Each cluster would hold sentences with the same syntactic pattern and similar semantics. This allows the user to search for answers to specific questions about the use of a medicine such as proper dosage quantity and frequency. This is done by specifying linguistic pattern and a list of concepts from the semantic ontology. For example, to inquire about proper dosage, some relevant concepts are "daily (temporal concept), take (healthcare activity), medicine (pharmacologic substance)" and so on. The evaluation was done by manually inspecting a number of such standardized questions, and qualitatively assessing if the tool was able to retrieve the answer. The method indeed provides meaningful results. One limitation within the work of Shafiee was the inability to detect nested linguistic constructs in one sentence.

3.2 Natural Language to Business Process Models

Business process models are utilized for process optimization, partial automation of the workflow, or documentation of the workflow implementation. Based on a comprehensive investigation in 2019 of the techniques, tools, and trends by Maqbool et al. [34], it was shown that current NLP techniques significantly simplify the process of BPMN model generation from textual re-

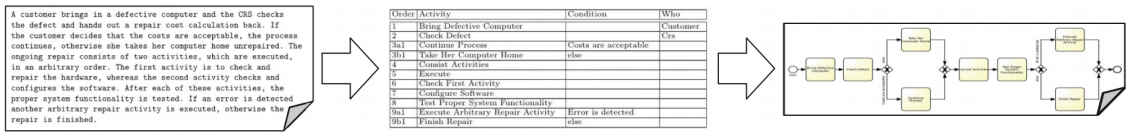

Figure 3.2: High-level methodology of Honkisz et al. [25]. Image taken from the same source.

quirements. This property implies that current techniques might be inadequate to be applied in current industries working with (real-time) systems. This section gives a brief description of recent methodologies for processing natural language into business process models.

Honkisz et al. [25] recently proposed a new method for business process model extraction from natural language texts. Their three-step methodology involves converting raw text into a spreadsheet representation, which is then used to generate an intermediate process model, as depicted in Figure 3.2. Their method applies both syntactic and semantic analyses (including POS tagging and dependency parsing) of the text to extract relevant and filter out irrelevant Subject-Verb-Object (SVO) constructs. Given the structured nature of these spreadsheets and, thus, the generated models, they allow for ease in checking its correctness and applying manual correction. These spreadsheets inspired us to create an intermediate representation that enables easy checking and correcting that is also capable of being translated into a formal output.

Pietikäinen [42] attempted to mitigate the workload of domain experts in the business processing field by constructing genre analysis methods and classifying the headings of the documents. These approaches would 'summarize' text documents into business process steps that are relevant for domain experts, which essentially saves time and effort.

Ferreira et al. [19] proposed a combination of NLP tools that work synchronously based on predefined mapping rules. These tools apply syntactic and logic analysis on input text to generate a rule-mapped text output that can easily be converted into a business process model.

Goncalves et al. [23] makes use of group narrative combined with text mining and natural language interpretation to automatically generate process models. Their approach involves tokenization as well as morphological, lexical, syntactic, and domain analysis of story texts to extract workflows. This approach generates a set of candidate models, of which a final model is manually picked by the user. As mentioned in the overview by Riefer et al. [46], it is a flexible approach, but it generates incomplete BPMN models.

All of the works described in this section perform syntactic analysis on the natural language input. Additionally, they take similar steps in their approach to extract process information from natural language. These steps (and their clear segregation as in [23]) inspired us in two ways. First, identifying the sub-tasks of our problem: lexical analysis, dependency parsing [25, 19], intermediate representation [25], and a final translation which differs among works. This helped us segment our design into five modules and inspired us to strive for a modular design. Furthermore, we benefited from individual ideas. Our idea of an intermediate representation was inspired by the intermediate model description of [25] which outlines the elements of a process in a readable yet formal representation. The use of mapping rules by [19] inspired us to create a similar set of rules mapping the intermediate representation to the final formal output.

3.3 Natural Language to Linear Temporal Logic

Linear temporal logic (LTL) is a well-suited technique for formalizing technical requirements, but is mathematically challenging to interpret. To simplify the creation of LTL for domain experts without a strong mathematical background, research is ongoing to develop approaches that translate natural language texts into LTL formulas, or vice versa. This section lists two important pieces of literature that cover this problem.

Brunello et al. [6] investigated a handful of current state-of-the-art methodologies concerning English-to-LTL translation problems. They state that current approaches attempt to extract time constraints from texts—they refer to this as "temporal expression recognition and normalization". Additionally, the challenge of defining a nesting hierarchy to an arbitrary depth as opposed to defining it to a certain fixed depth. They conclude from the literature they reviewed that a solution to translate unbounded natural English text to unbounded LTL formulas is missing.

Sànchez-Ferreres et al. [61] recently developed a framework that annotates textual descriptions both manually and automatically such that these can be transformed into temporal formulas, allowing for a reasoning mechanism to produce or validate processes. Their formal model is used to translate specially annotated text, named ATDP, into LTL. The model is able to capture ambiguity within text and to separate phases of a process using a recursive approach combined with the annotated data and formal semantics.

We find out from [4, 6] that other analysis techniques that involve grammatical constraints [69] and regular expressions [53] are not as suitable to detect nesting. Albeit theoretically possible, it requires a tedious effort due to the exponentially increasing number of variants a grammatical construct can appear in. This requires many more rule variants to be specified in comparison with the flexibly-matched dependency patterns of [61]. To illustrate, consider the sentence:

> If you take the medicine before you have breakfast, and your last meal was more than eight hours before, take it with water and antacid or milk.

This sentence contains two *before* conjunctions and *if*, *and* and *or* conjunctions. Furthermore, the second *before* has an implicit conjunct, i.e., the *before*-clause is not explicitly stated. These complications cause linear approaches, as [53, 69], to struggle in an attempt write rules for cases this specific. For that reason, we chose to utilize dependency parsing in our approach.

3.4 Process Extraction from Text

Bellan et al. [4] recently performed a qualitative analysis on state-of-the-art approaches and tools for process extraction from text to give insights on current limitations. They state that current state-of-the-art papers mostly rely on CREWS intermediate representation. Additionally, they define process extraction from text as the problem of generating a process model diagram from its procedural description using function f, which can be split up in sub-functions f_a, *text-to-world*, and f_b, *world-to-model*. These sub-functions can be broken down into smaller parts, namely:

- $f_{a,i}$: resolving anaphoric references
- $f_{a,ii}$: filtering uninformative textual fragments

- $f_{a,iii}$: extracting process elements and structures from text
- $f_{b,i}$: adding process elements and structures to correctly create process models
- $f_{b,ii}$: connecting process elements together using similar logic conveyed in text
- $f_{b,iii}$: generating textual labels for each process element and generate the process model diagram

Out of these tasks, we identified that our problem is, to some extent, related to $f_{a,iii}$ and to $f_{b,ii}$. Task $f_{a,iii}$ is concerned with extracting process elements and process structures. Elements of a process are activities, roles, events, and so on, while process structures are gateways that control the flow of a process [65]. While we do not attempt to extract process elements, we do extract the relation(s) between two or more process fragments within a sentence, which contributes to discovering a process structure.

Task $f_{b,ii}$ is concerned with connecting process elements together with connectors that are formally defined in process semantics, such as the sequential connector which simply declares that an activity A is sequentially followed by an activity B. Such a connector is derived based on its formal logic being similar to the logic conveyed in the text, e.g., through a sentence as "After A, B". While we do not detect connectors between process elements such as between an agent and its role or an activity and its resource, we do detect certain connectors between activities such as the ones conveyed by *after* and *before*.

Identifying these two tasks helped us distinguish them from each other. Extracting process elements and structures through syntactic and semantic analysis is part of the text-to-world function f_a, while projecting a formal semantics on a linguistic construct is part of a different function with a different set of tasks, the world-to-model function f_b. This further backed the significance of an intermediate representation (inspired by works covered in Section 3.2) that mediates between these two functions.

Finally, according to Bellan et al. [4], current state-of-the-art contains three main limitations, namely (1) techniques are highly tailored to specific forms of input data; (2) datasets are used that do not represent solid benchmarks; and (3) metrics are used that are not refined to the specific problem. These are relevant limitations to consider when developing a methodology aimed at solving this problem.

3.5 Annotation of Procedural Texts

Quishpi et al. [45] developed an open-source tool that extracts process information from text (e.g., actions, events, agents, and roles), and uses that information to annotate the text, making it easier to read and navigate through. In addition to increased readability and navigability, their approach also makes the processed data accessible for machine learning applications and formal reasoning.

They claim that annotated textual descriptions of processes allow for formal reasoning or simulation of the underlying process described in the text document. Related techniques require text that is manually annotated in ATDP format [61], which they aim to automate using an NLP tool

that finds the dependency tree of a sentence. The dependency tree is then matched to certain patterns in a flexible manner. This flexibility matching is claimed to capture varying dependency trees that all describe the same process elements. The extracted ATDP is then exported to a model checker.

Sanchez-Ferreres et al. [62] integrate the work of Quishpi et al. [45] to bridge the gap between process mining and ATDP specifications. They do so through a simulator capable of generating an event log based on the ATDP specifications. This enables techniques such as process discovery and conformance checking.

As pointed out in [45, 61, 62], dependency parsing is a suitable text analysis technique for detecting nesting within a sentence.

3.6 Formalization of Software Requirements

In this section, we review works dealing with the formalization of software requirements.

3.6.1 Software Design Documents to Requirements

Wein et al. [73] recently proposed a solution to automatically extract software requirements from functional design documents. Requirement extraction is performed in three steps:

1. Extracting sentences from the document, which is done using keyword and base verb extraction.
2. Incorporating coreferent text, which is done to tackle ambiguous sentences that were found during semantic analysis and checks on ambigouos keywords and parts-of-speech.
3. Aligning the extracted text to the (guidelines of the) official software requirements, which is done to assure the relevant words are present in the extracted sentences to match the guidelines.

They have shown that this approach can yield promising results, providing evidence of the usefulness of (automated) NLP techniques within specifying formal requirements based on texts.

3.6.2 Informal Specifications

There are also documents that contain informal specifications (software requirements) that require to be specified formally for automated processing. Ghazel et al. [21] developed a tool that allows for formalization of informal specifications via an iterative process which relies on a set of basic refinement patterns. Applying these refinement patterns iteratively will eventually create logical Computation Tree Logic (CTL) formulas.

Their approach, however, does not allow for full formalization, but only eases manual formalization after their method is executed. Additionally, their work is unable to address the ambiguity that is present in the word *or* in natural language, as this could imply either OR or XOR in logic. They state that the distinction is difficult to make automatically, and that manual labelling is required for this word to match its logical equivalent.

3.7 Tools

For this project, we used Python since many tools and libraries are available for NLP in this programming language. In this section, a brief summary of the widely-used libraries is presented.

NLTK[1] (Natural Language Toolkit) is quite a popular NLP library for Python that was developed by the University of Pennsylvania. NLTK is essentially a string processing library. We use it partially in our Module 4. The tasks it provides are summarized in Figure 3.3.

spaCy[2] is yet another popular library that is growing fast in the field of NLP. This library boasts about its speed and accuracy with publicly available benchmarking means, which beats other frameworks. A comparison of speed and accuracy with other prominent libraries is given in [56]. spaCy is also covered in the comparison of Figure 3.3 with a slight modification required to that table; in fact, spaCy released a co-reference resolution module recently in late May of 2021 [55]. spaCy is our main tool of choice for this thesis. It was first released in 2015.

Stanford CoreNLP Python[3] is a Python library that uses the well-known *CoreNLP*[4] library. It was developed by Stanford's Natural Language Processing Group[5] which was first released in 2010.

FreeLing is another strong contender. It includes all of the features shown in Figure 3.3 and was developed by the Polytechnic University of Catalonia (UPC) from the same group behind the work of Quishpi et al. [45] (Section 3.5). This tool was the first choice before we switched to spaCy. The reason FreeLing fell out of favor was the technical difficulties faced with the installation, configuration and usability, especially compared to spaCy.

We attempt to assess where spaCy's parser is placed in relation to others. spaCy's dependency parser uses the Clear dependency set[6] [7]. The Clear set is not the latest. The latest dependency set is the Universal Dependency Relations (UD version 2) set [66]. This may indicate that spaCy's parser does not represent the state-of-the-art. However, spaCy makes the claim that it does in fact house state-of-the-art speed and accuracy [55]. We can back that claim for the following reasons:

- Adopting a dependency set requires tedious work and major changes to the parsing algorithm [9].
- The state-of-the-art dependency set has not yet been implemented [58].
- The Clear set is a modification of Stanford's Universal Dependency version 1 set [7, 10], aimed for increased speed and accuracy.

[1] http://www.nltk.org/
[2] https://spacy.io/
[3] https://github.com/stanfordnlp/python-stanford-corenlp
[4] https://stanfordnlp.github.io/CoreNLP/
[5] https://nlp.stanford.edu/
[6] https://www.mathcs.emory.edu/~choi/doc/cu-2012-choi.pdf

	SPACY	NLTK	CORENLP
Programming language	Python	Python	Java / Python
Neural network models	✓	✗	✓
Integrated word vectors	✓	✗	✗
Multi-language support	✓	✓	✓
Tokenization	✓	✓	✓
Part-of-speech tagging	✓	✓	✓
Sentence segmentation	✓	✓	✓
Dependency parsing	✓	✗	✓
Entity recognition	✓	✓	✓
Entity linking	✓	✗	✗
Coreference resolution	✗	✗	✓

Figure 3.3: Comparison of functionalities offered by spaCy, NLTK and CoreNLP. (Source: [64])

Chapter 4

Solution Approach

To solve the problem of this thesis, we developed a framework to transform procedural descriptions found in natural language text into a number of formal representations. We implemented the tool in Python assisted by components from the NLP framework spaCy. We first give a brief description of the methodology followed in the solution. We then explain the modules of our solution in Sections 4.1 to 4.5. Next, we discuss the observed limitations and threats to validity in Section 4.7.

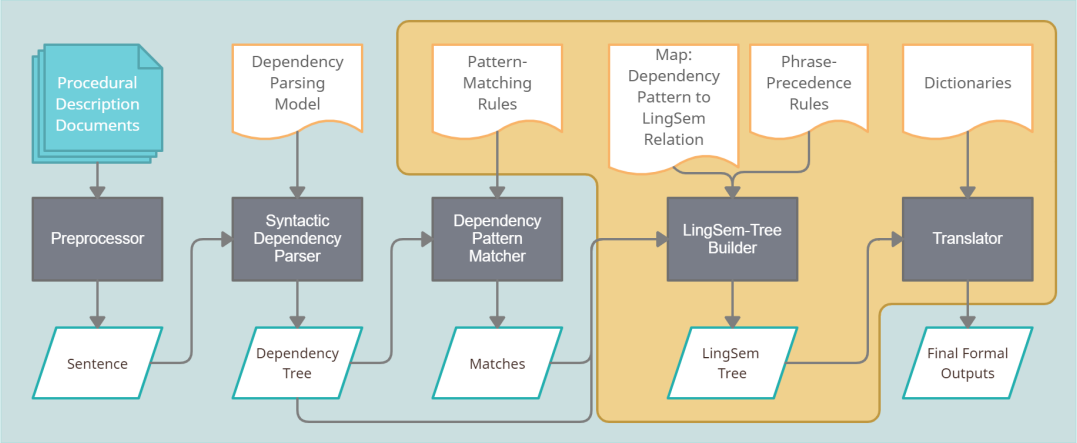

Figure 4.1: Flowchart showing the high-level design of the tool with the contribution of this thesis highlighted.

An overview of the approach is visualized in Figure 4.1, consisting of five modules. Briefly explained, Module 1, the preprocessor, prepares the input text by segmenting it into sentences, and applying a POS tagging pipeline on each sentence. Module 2 takes a sentence as input, applies dependency parsing utilizing a trained model, and produces a dependency tree such as the one in Figure 4.2. Module 3 matches certain predefined patterns against the tree and outputs the matches. Module 4 is explained in two tasks. Task 1 combines the tokens from the dependency

tree and forms them into predicates. To elaborate, tokens that have a common head/ancestor in the dependency tree that are not part of any match, but the ancestor is, are grouped to form a predicate. Task 2 transforms matches, which are now completed with predicates, into subtrees of linguistic-semantics (*LingSem*) relations, and arranges those in a semantic hierarchy, we call a LingSem tree, based on the given dependency hierarchy. Finally, Module 5 translates a LingSem tree into one or more formal languages. The current implementation supports a small set of constructs from LTL. The output is then an LTL expression.

In the following sections, the inputs, outputs and innerworkings of each module is explained. We demonstrate that by applying the modules on a running example, displaying the output at each step.

Sample Input Throughout the implementation work, we have worked with sample input text of 360 words in 18 sentences describing 'how to use Ibuprofen oral'. The text is part of the WebMD online medical resource and can be viewed online[1]. This text is used for the purpose of evaluation in Chapter 5.

The sentences of this text are shown in Table 5.1.

We chose a relatively complex sentence from the text, precisely the second sentence $\mathcal{S}_2$ (with a slight modification for brevity). This sentence is our running example through the rest of the modules:

> $\mathcal{S}_2$: If your doctor has prescribed this medication, read the Guide before you start taking ibuprofen and each time you get a refill.

We consider this a representative sentence because it contains an ordering using *before*, a decision/conditional using *if*, and a nesting, particularly, the nesting of the *before* construct within the *if* one.

4.1 Module 1: Preprocessor

Goal: To apply preprocessing to the input text and prepare it for further analysis.

Input: Texts of the procedural description type.

Output: Texts split into sentences and sentences into tokens. Each token is annotated with a POS tag. Since there are no operations across sentences for the next modules, we consider a single sentence as the output of this module and the input of the next.

Module 1 contains a pipeline that is applied to the input text. First, it segments the sentences. Then, it tokenizes each sentence into tokens, and performs lexical analysis to attribute each token with a Part-of-Speech (POS) tag [67]. This pipeline is composed entirely of functions made by spaCy.

The output of this module for Sentence $\mathcal{S}_2$ is shown in Table 4.1. The output is configured to truncate noun phrases, hence phrases such as 'your doctor', 'this medication' and 'the Guide' are

[1] https://www.webmd.com/drugs/2/drug-5166-9368/ibuprofen-oral/ibuprofen-oral/details

Token	POS Tag	Tag Meaning
If	SCONJ	Subordinating conjunction
your doctor	NOUN	Noun
has	AUX	Auxiliary
prescribed	VERB	Verb
this medication,	NOUN	Noun
read	VERB	Verb
the Guide	PROPN	Proper Noun
before	ADP	Adposition
you	PRON	Pronoun
start	VERB	Verb
taking	VERB	Verb
ibuprofen	NOUN	Noun
and	CCONJ	Coordinating conjunction
each	DET	Determiner
time	NOUN	Noun
you	PRON	Pronoun
get	VERB	Verb
a refill.	NOUN	Noun (Noun phrase)

Table 4.1: Output of Module 1 for the sample sentence $\mathcal{S}_2$.

displayed as a single token.

One thing to notice in the sample output is that *ibuprofen* is tagged as a noun rather than as a proper noun. Once the first letter is capitalized, the module tags it as a proper noun. Although such an error can propagate and affect the outputs of next modules, this particular case does not.

4.2 Module 2: Syntactic Dependency Parser

Goal: To perform dependency parsing on each sentence and find dependency relations between its tokens.

Input: A sentence.

Output: The sentence with dependency relations between its tokens, i.e., a dependency tree.

Required Resource: A dependency parsing model.

Module 2 takes the annotated sentence as input and uses spaCy's dependency analyzer/parser to produce a dependency tree. As pointed out in Section 2.1.2, the precise formalism of a dependency tree depends on the algorithm used and the dependency set (set of dependency relations). spaCy uses the more accurate and more advanced graph-based dependency parsing and the Clear dependency set.

Sample Output

The output of Module 2 for our running example $\mathcal{S}_2$ is a dependency tree shown in Figure 4.2. The figure shows the tokens of the sentence in their original order from left to right, with the dependency relations above and the POS tag of each token below it. Every directed edge specifies

Label	Meaning	Label	Meaning
advcl	Adverbial-clause modifier	advmod	Adverbial modifier
aux	Auxiliary	cc	Coordinating conjunction
ccomp	Clausal complement	conj	Conjunct
det	Determiner	dobj	Direct object
mark	Marker	npadvmod	Noun-phrase adverbial modifier
nsubj	Nominal subject	nummod	Numeric modifier
pcomp	Prepositional complement	prep	Prepositional modifier
relcl	Relative clause modifier	xcomp	Open clausal complement

Table 4.2: The dependency-relation labels shown on various dependency trees throughout the thesis and the full-form term of each.

a dependency relation carrying a certain label. The labels shown and their meanings are given in Table 4.2. The complete set of dependency relation labels is explained further in Section 4 of the CLEAR style manual[2] [7].

Since the produced tree in Figure 4.2 has a mistake—discussed at length among threats to validity in Section 4.8.1—, for the next modules, we use a corrected dependency tree shown in Figure 4.4.

The Parsing Model

This module, the dependency parser, requires a dependency parsing model as an essential resource to perform the parsing. A parsing model is generated through training on a large dependency-labeled corpus. The model, provided by SPACY, is named `en_core_web_trf` and it aims for accuracy over efficiency [57].

The model boasts an accuracy for unlabelled dependency relations of 95% and labelled dependency relations of 94%. Therefore, 5% of the relations formed are expected to be wrong. This is a limitation that is expected to abate as dependency parsing models advance. For now, it requires human intervention to bypass.

Design Decision: Using a Dependency Parser

The decision to use a dependency parser is based on the need to capture the nesting of linguistic constructs. The analysis in Section 2.1.2 shows that a syntactic dependency tree captures nesting down to an arbitrary depth, given sufficient computational resources. This decision is also inspired by related works such as [45].

The information we aim to extract from a dependency tree is: (1) how clauses or predicates of a sentence relate to one another, and (2) what the hierarchy of dependence is among these clauses. There are many words in the language that conjunct two clauses in a sentence such as *if*, *before*, *when*, *while*, *during*, etc. Furthermore, any one conjunct may also have two clauses in conjunction, any of which may in turn be comprised of two clauses in conjunction. This nesting may extend to an arbitrary depth. We are interested in capturing the hierarchy of connections of clauses.

[2] https://www.mathcs.emory.edu/~choi/doc/cu-2012-choi.pdf

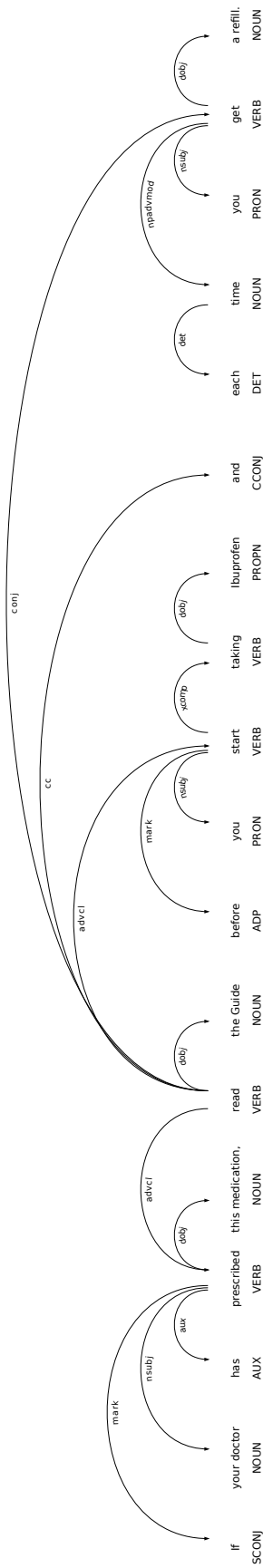

Figure 4.2: Dependency tree of $\mathcal{S}_2$; unmodified.

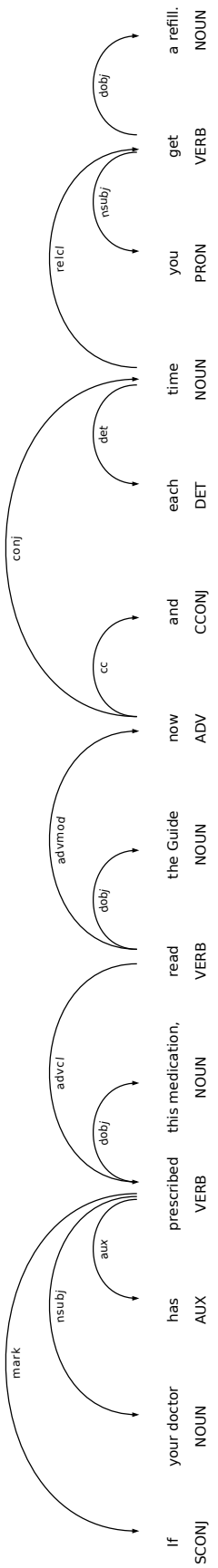

Figure 4.3: Dependency tree of $\mathcal{S}_{2_1}$ which is $\mathcal{S}_2$ simplified by replacing the *before* adverbial clause by the adverb *now*.

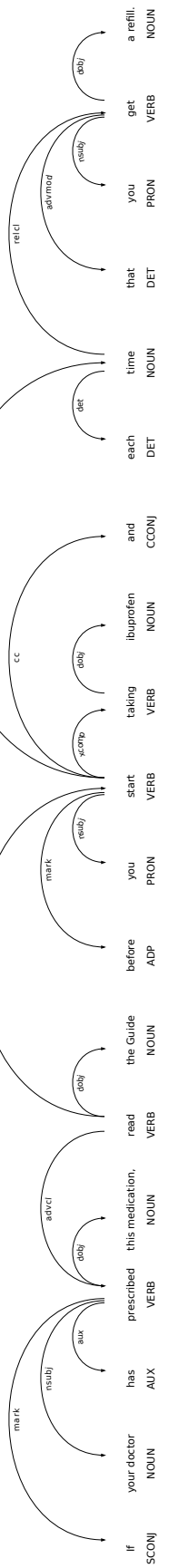

Figure 4.4: Dependency tree of $\mathcal{S}_{2_2}$, which is $\mathcal{S}_2$ modified by adding the disambiguating complementizer *that*.

4.3 Module 3: Dependency Pattern Matcher

Goal: To find certain pre-defined patterns of dependency relations in a sentence.

Input: A dependency tree, and a set of pattern-matching rules housing the dependency patterns to find in the dependency tree.

Output: A set of matches for each sentence for each pattern.

The role of Module 3 is to receive the dependency trees which Module 2 outputs, and search them for subtrees of interest, i.e., parts of the dependency tree which carry elements of natural language we believe are indicative of the semantics sought. This search is carried out by a pattern matching engine that is part of spaCy. Thus, Module 3 is, in fact, spaCy's pattern matcher. A search query made to the pattern matcher is called a *dependency pattern*. In the following subsection, we define what a dependency pattern is, provide an example, and explain the procedure we followed to construct a dependency pattern based on some semantics of interest.

4.3.1 Dependency Patterns

A dependency pattern is essentially a dependency tree with constraints on its nodes and edges. A node with variable constraints on it is called a *constrained node*. Likewise, an edge with variable constrains on it is called a *constrained edge*. A dependency tree is a tree (N_c, E_c) where is N_c is a set of constrained nodes and E_c is a set of constrained edges. It acts as a tree search query. For a node in N_c, there are two constraints expressed in the form of the following two sets:

- A set of POS tags, such that the constrained node matches any token whose POS tag is in the specified set.
- A set of string patterns, typically specified by regular expressions. A matching token must satisfy one of the specified string patterns.

A token must satisfy both these constraints in order to count as a match to the constrained node. Additionally, one or both of these constraints can be dropped.

An edge constraint specifies a dependency relation and consists of the following three elements:

1. a source node $n_s \in N_C$,
2. a destination node $n_d \in N_C$ where $n_d \neq n_s$,
3. and a set of dependency relations, such that a matching edge must satisfy one of the dependency relations in the set.

An edge in the queried dependency tree must satisfy all three elements of the constraint in order to count as a match to the constrained edge.

Given a dependency tree, called the subject tree, and a dependency pattern, the problem of finding matches of the dependency pattern within the subject tree is known in the literature as tree pattern matching [8, 24] and is defined as follows. Given a pattern tree P consisting of k nodes labeled $v_1, \ldots, v_k$, and a subject tree S, we say that P matches S at node n (i.e., the root node of P

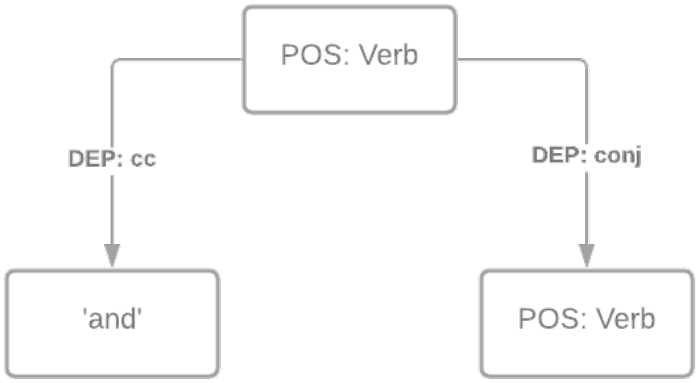

Figure 4.5: The dependency pattern specified to detect the *and* conjunction of two verbs.

matches a node n in S) if there are k subtrees of S labeled $t_1, \ldots, t_k$ such that substituting t_i in v_i for all i in $\{1, \ldots, k\}$ yields the subtree of S whose root is n.

In our framework, dependency patterns are defined following the Semgrex syntax created by the Stanford NLP Group and used by SPACY [60, 37, 59]. We define them in an external source (a JSON file) shown in Figure 4.1 as 'Pattern-matching Rules'.

An example of a dependency pattern is shown in Figure 4.5. This tree consists of one parent and two child nodes. The parent carries one constraint specifying the POS tag matching it to be the `VERB` tag. Then, a matching token must have two children according to the pattern, one carrying the dependency `conj` and the POS tag `VERB` and another carrying the dependency `cc` and matching the string 'and'. This pattern is meant to detect any two verbs in conjunction through *and*.

A second example of a dependency pattern can be expressed simply as follows: `POS:VERB` $\xrightarrow{advcl}$ `POS:VERB` $\xrightarrow{mark}$ *String:'before'*. To explain the notation briefly, every $\xrightarrow{dep\_label}$ represents a dependency relation carrying a label and connecting two constrained nodes. In this example, the first and second nodes are constrained by the POS tag `VERB`, whereas the third carries a string 'before' to be matched exactly. As a shorthand notation, we discard the names of the constraint specifiers (POS and String) since they can be inferred by the reader, thus writing the pattern as: `VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ *before*.

Examine the output of Module 2, which is the dependency tree in Figure 4.4, to find the aforementioned pattern within it. Indeed, we see that the subtree *read* $\xrightarrow{advcl}$ *start* $\xrightarrow{mark}$ *before* matches the pattern.

In SPACY, this dependency pattern is specified in the JSON format as a collection of name-value pairs [72]; and it is shown below in Listing 4.1:

```
1  {
2    "advcl_before_pattern" : [
3
4      # Token 1: main-verb.
5      {
6        "RIGHT_ID": "main-clause verb",
7        "RIGHT_ATTRS": {"POS": "VERB"}
8      },
9
10     # Token 2: advcl-verb, with relation: main-verb --advcl-> advcl-verb.
11     {
12       "LEFT_ID": "main-clause verb",
13       "REL_OP": ">",
14       "RIGHT_ID": "adv-clause verb",
15       "RIGHT_ATTRS": {"DEP": "advcl"}
16     },
17
18     # Token 3: The clause marker 'before', with relation advcl-verb --mark-> 'before'.
19     {
20       "LEFT_ID": "adv-clause verb",
21       "REL_OP": ">",
22       "RIGHT_ID": "adv-clause mark",
23       "RIGHT_ATTRS": {"DEP": "mark", "LOWER": "before"}
24     }
25   ]
26 }
```

Listing 4.1: The `adv-clause-before` pattern specified in spaCy's syntax.

To understand the spaCy-formatting, keep in mind the simple notation of the same pattern: `VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ *before*. The spaCy-formatted pattern above is specified primarily through the three nodes (lines 4, 10 and 18), where each node represents a token. Each token is further specified through a set of attributes contained in the field called `RIGHT_ATTRS`, shown at lines 7, 15 and 23 for each of the three tokens respectively. The attributes in `RIGHT_ATTRS` can specify the POS tag of the token as in line 7. It can specify the string of the token as in line 23. It can also specify a dependency relation between two tokens, the ones whose labels are specified by `LEFT_ID` (line 12) and `RIGHT_ID` (line 14), with the direction and nature of the relation specified by an operator in `REL_OP` (line 13). In this case, the relation denoted by `>` is a direct child relation and its direction is from the `LEFT_ID` node to the `RIGHT_ID` node. A detailed explanation is available on spaCy's online documentation[3].

Requesting the matcher to find this pattern within the tree of Figure 4.4 returns a successful match shown in the listing below.

```
Name of pattern matched: advcl_before_pattern.
```

[3] https://spacy.io/api/dependencymatcher

Dependency Pattern Name	Dependency Pattern Specification
`adv-clause 'before'`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ 'before'
`advmod 'before'`	`VERB` $\xrightarrow{advmod}$ *before* $\xrightarrow{pcomp}$ `VERB`
`prep 'before'`	`VERB` $\xrightarrow{prep}$ *before* $\xrightarrow{pcomp}$ `VERB`
`adv-clause 'if'`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ 'if'
`adv-clause 'when'`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ 'when'
`adv-clause 'unless'`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ 'unless'
`adv-clause 'after'`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ 'after'
`adv-clause 'until'`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ 'until'
`adv-clause 'while'`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ 'while'
`adv-clause generic`	`VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ *
`negation`	`VERB` $\xrightarrow{neg}$ `PART`
`and-conj`	(`VERB` : t_1 $\xrightarrow{cc}$ 'and') $\wedge$ (t_1 $\xrightarrow{conj}$ `VERB` : t_2)
`or-conj`	(`VERB` : t_1 $\xrightarrow{cc}$ 'or') $\wedge$ (t_1 $\xrightarrow{conj}$ `VERB` : t_2)

Table 4.3: The dependency pattern matching rules defined in our framework.

```
Tokens matched:
  main-clause verb: read
  adv-clause verb: start
  adv-clause mark: before
```

Listing 4.2: Result of matching the `advcl-before` dependency pattern against the tree in Figure 4.2.

The current implementation has several rules defined in such a way. Table 4.3 shows the label of each rule and the dependency pattern it is meant to match side by side.

Next, we explain the procedure we followed to arrive at these pattern specifications.

4.3.2 The Procedure Followed for Crafting a Dependency Pattern

The previous example shows one of the patterns we use to detect and identify two predicates in a sentence related via *before*. Here, we show the procedure we follow in order to arrive at such patterns. Briefly, we look at individual templates that were created by Dwyer et al. [12, 13, 14] and deemed frequent by the process modeling community. Then, for each template, based on our understanding, we infer one or more linguistic patterns that may express the same semantics as the process template. Finally, based on understanding again, we infer the equivalent LTL formula. Partial results of this procedure are shown in Table 4.4. In the following, we explain the procedure in more detail.

We start by targeting a certain semantics we aim to detect within natural language. Since natural language carries a vast range of semantics, we focus on a set of semantics that better serves our purpose. The Dwyer set of process templates (Section 2.3.3) is suitable because:

1. It is a much narrower set of semantics than natural language, which reduces the complexity of our task.
2. It was adopted by the process modeling community to describe process model templates; the studies sprawling the language of DECLARE [41] also studied the frequency and significance of these templates.
3. It was made out of frequently observed patterns within software design texts which, when compared to procedural descriptions, we assume is close enough to serve the purpose of choosing an initial set of semantics to aim for.

Another set of semantics is found through common or desired LTL expressions such as $x \ \mathcal{U} \ y$.

Next, we focus on finding the linguistic pattern(s) that express the desired semantics. a simple example is the aforementioned LTL expression. Using our understanding of language and of LTL, we can make the assumption that the linguistic pattern *x until y* carries the semantics of the LTL expression $x \ \mathcal{U} \ y$. In the same way, Shafiee [53] used her understanding next to observing her case study text to associate certain linguistic patterns with formal expressions. In Table 4.4, we list some of these findings, the process templates of interest along side the LTL equivalent of each and one linguistic pattern which may express the process template.

Process template	A linguistic equivalent	LTL semantics
Response(x, y)	If x, then y.	$\mathcal{G}(x \rightarrow \mathcal{F} \ y)$
Precedence(x, y)	x before y.	$\mathcal{G}(\neg y \ \mathcal{W} \ x)$

Table 4.4: Some of the semantics of interest for Module 3, expressed in process templates, one linguistic equivalent for each and the equivalent LTL formula.

Take the template of *Precedence*(x,y). We know from [53] that one of its linguistic manifestations is *x before y*. We observe the dependency tree in Figure 4.4 and notice the relation between the main clause and the before-clause including the *before* function word. That relevant subtree is drawn as: *read* $\xrightarrow{advcl}$ *start* $\xrightarrow{mark}$ *before*. We generalize this subtree by replacing the non-key words with their POS tags arriving at the `advcl-before` pattern we have seen earlier. However, this is not the only dependency pattern carrying the desired semantics of *before*. To find out more, we resort to the method of scanning the case study text as explained next.

Scanning a corpus to find variant dependency patterns

One way to find more dependency patterns to cover is through observing more dependency trees to see different forms of how a function word of interest relates to other tokens in a sentence. The exact procedure we followed for this method is as follows:

1. Prepare a corpus such as the collection of articles we acquired from WebMD.
2. Run the corpus through Module 1 and Module 2, thereby segmenting it into sentences, POS tagging each token and producing a dependency tree for each sentence.
3. Pick a keyword of interest. In our experiment, we picked the word *before*. In addition,

have at least one dependency relation of interest that is believed to convey a certain sought meaning.

4. Craft a simple dependency pattern consisting of the keyword, but excluding the already known relations.
5. Let the dependency matcher (Module 3) find matches in the corpus for the pattern specified.
6. Pick an arbitrary set of matched dependency trees for manual inspection. Upon inspection, the following scenarios may occur:
    - A new dependency pattern is found that conveys the same sought meaning. In this case, add it to the already known relations of Step 3, and repeat the process starting from Step 4.
    - If no new patterns are found, end the procedure.

This simplistic procedure may lead to dependency patterns being added to the set of interest that convey the sought meaning sometimes but convey a different meaning other times. The decision of whether to include such patterns or not should then be decided after collecting a statistic on the frequency of each meaning conveyed. Computing such statistic through manual inspection of an entire corpus is tedious work and was not part of this project.

We applied this procedure for the function word *before* to find patterns other than the `adv-clause before` pattern that would convey the same meaning. We manually inspected 68 sentences of the WebMD corpus that had at least one use of the word *before*, and found the following:

- 56% of the time, the word *before* appeared as the mark word of an adverbial-clause modifier, as in Figure 4.2 where the clause is "before you start taking Ibuprofen".
- 8% of the time, it appeared as an adverbial modifier (`advmod`) itself (as opposed to a clause's mark). This relation appears in sentences as "The doctor may inquire if the symptoms were noticeable before", or simply "It was here before". Notice that there is no *before*-clause in this example.
- 34% of the time, it appeared as a preposition. This occurs in sentence like "Call before your appointment" where *before* is followed by a noun or a noun phrase. It also occurs when before is followed by a verb, especially in its gerund form, e.g., "Consult the guide before visiting the pharmacist".

By following this procedure, we were able to see the necessity of adding two sister patterns to the `adv-clause before` pattern. While the `adv-clause before` pattern is `VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ *before*, the two new sister patterns are as follows:

1. The `advmod before` pattern: `VERB` $\xrightarrow{advmod}$ *before* $\xrightarrow{pcomp}$ `VERB` .
2. The `prep before` pattern: `VERB` $\xrightarrow{prep}$ *before* $\xrightarrow{pcomp}$ `VERB` .

An input sentence may contain more than one match from one or more rules. These matches are stored and forwarded to Module 4, which distinguishes them from each other through their labels.

Focusing on relations between verbs rather than between nouns.

All of the dependency patterns defined specify relations between two verbs, rather than between a verb and a noun or two nouns. There are two reasons for this. First, verbs represent activities, and activities are of interest because of our pre-context of extracting process fragments—which are identified by relations between activities. Second, a verb is the head of the dependency subtree representing a clause or predicate. In other words, a verb is the center and identifier element of a clause or a predicate. On the other hand, a noun could be an adjective, a subject, an object, or a prepositional complement, i.e., grammatical roles all of which are not considered the central identifier element of a clause or a predicate, nor do they represent activities [2].

A limitation of POS tagging and dependency parsing in capturing activities.

As explained above, while crafting dependency patterns, we rely on the fact that activities are represented by verbs. However, the flexibility in the use of natural language sometimes leads to sentences where verbs are hidden—thereby making activities implicit rather than explicit. As an example, consider hiding the verb *helping* in the sentence: "Help yourself before [helping] others". In such sentences, the analysis techniques of POS tagging and dependency parsing fail at identifying the implicit activity, and consequently fail at detecting a relation between activities, as elaborated below through the example. This is a limitation of the two natural language analysis techniques we use in this thesis. The limitation can be mitigated through deploying more advanced dependency parsing models that go beyond language-based analysis into utilizing the context of a sentence and the knowledge it carries [49].

This limitation is most evident in the `prep-before` pattern: `VERB` $\xrightarrow{prep}$ *before* $\xrightarrow{pcomp}$ `VERB`. The last token being a verb means this pattern does not match sentences as "Take one dose of the medicine before dinner". The word *dinner* represents an implicit activity in this sentence, yet is not detected as such because it is a noun. Another example is the aforementioned "Help yourself before [helping] others". Observe its dependency tree in Figure 4.6. Here, the prepositional complement is the noun *others*. The currently used dependency parser has no way of distinguishing a noun prepositional complement representing an activity from one that does not. Take as a contrary example the sentence: "Stand still before your physician" and its dependency tree in Figure 4.7. Here, the noun phrase *your physician* is not related to an activity, yet the dependency relation is the same as with the word *dinner* (Figure 4.8) which does represent an activity. In all these examples, our human knowledge of context is what resolves the ambiguity, i.e., knowing that dinner represents an activity; and knowing that standing before a physician means performing the activity of standing in front of an agent, called the physician, rather than executing it prior to the agent's own execution. Therefore, as context-utilizing dependency parsers become available, this limitation should be resolved [49, 48].

Design Decision: The rules being sparse instead of amalgamated.

Observing Table 4.3, the reader may wonder why many similar patterns exist separately, such as the `adv-clause` patterns, if they can be combined into one generic `adv-clause` pattern. During the implementation, we decided to combine similar rules under one definition for the sake

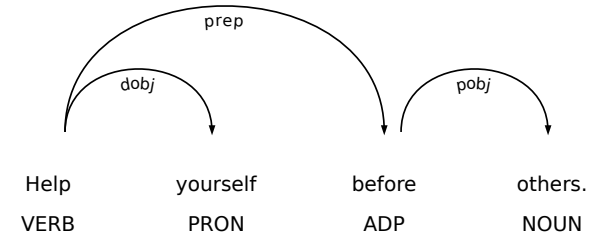

Figure 4.6: Dependency tree of "Help yourself before others".

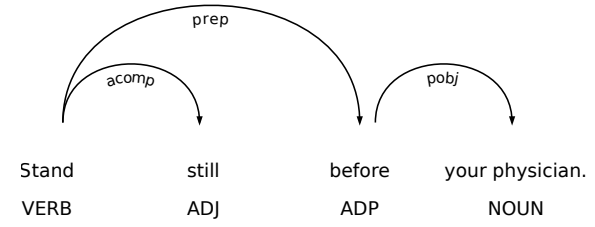

Figure 4.7: Dependency tree of "Stand still before your physician".

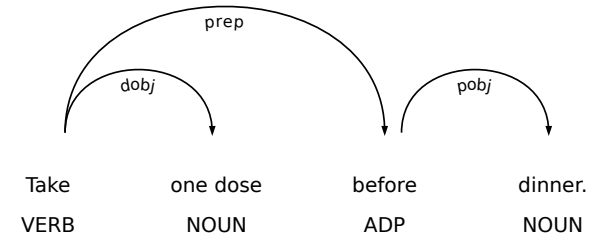

Figure 4.8: Dependency tree of "Take one dose before dinner".

of modularity. For example, all the `adv-clause` rules have exactly the same structure. They all match the pattern `VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{mark}$ [some function word]. They only differ in the function word that each rule matches, i.e. *if*, *before*, *when*, etc. It was possible to make one rule that matches against all different adverbial clauses or more precisely against adverbial clauses with a list of specific mark works. So we created one rule combining all adverbial clauses and labeled it `adv-clause generic`. However, soon after implementing that, we found it to be counter-productive for the purpose of the tool and how the next modules use the matches. Every mark word carries different semantics. The semantics of *if* is different from that of *before* and so on. Combining these marker words under one rule means that the matches found for the combined rule will all have the same label (`adv-clause generic`) attached to them. Since our goal is to convert the input into formal expressions, it is essential to know exactly which semantics is the one carried by a certain match, and so it is essential that each match carries a label that uniquely describes the semantic meaning it withholds.

4.3.3 Output of Module 3

The output of the dependency matcher to our running example $\mathcal{S}_2$, is a list of matches as follows:

```
Sentence 2: If your doctor has prescribed this medication, read the Guide before you start
    taking ibuprofen and each time that you get a refill.
  Match 1, Rule 4 (adv-clause 'before' pattern):
    main-clause verb: read
    adv-clause verb: start
    adv-clause mark: before
  Match 2, Rule 7 (adv-clause 'if' pattern):
    main-clause verb: read
    adv-clause verb: prescribed
    adv-clause mark: If
  Match 3, Rule 8 (and-conj-pattern):
    clause-1-verb: start
    clause-2-verb: get
    conj-particle: and
```

4.4 Module 4: Linguistic Parse Tree Builder

Goal: To construct predicates and to give hierarchy to the dependency matches.

Input: A map from dependency matches to LingSem relations, and a set of precedence rules.

Output: A LingSem tree, i.e. a tree of LingSem relations; this is our lossless intermediate representation.

Module 4 plays the central role in the formalization of a sentence. The input to Module 4 is the dependency tree of the input sentence and the matched patterns of interest. The module uses two resources: the dependency-to-LingSem map and a set of rules to specify precedence among LingSem relations. The module carries out two tasks in order to produce its intended output. The

first task is the construction of predicates from the dependency tree and the matches. The second task is to arrange these predicates into a special-purpose parse tree we call a LingSem tree—a formal intermediate representation that is lossless, human-readable and easy to automatically transform into the final formal output. The format of the LingSem tree will become clear in what follows. We first provide an overview before explaining each task in detail.

Since the goal of this project is to capture process-related semantics into formal expressions, we defined in Module 3 dependency patterns with the purpose of detecting such semantics. These are patterns such as `advcl-before`, `advcl-if` and `and-conj`, evident in the sample input of Figure 4.4. Each of these patterns was defined earlier in order to capture a semantic function of language. We call such linguistic semantics functions *LingSem relations*. More precisely, a LingSem relation expresses a connection between two predicates of a sentence through a linguistic function word such as *before*, *if* and *and*. One of the two predicates occurs in the main clause while the other occurs in the dependent/subordinate clause. We chose the notation of a LingSem relation to be as follows:

LingSem relation notation: *function-word* (*sub-clause*, *main-clause*)

The dependence of one clause on another is derived from the dependency tree produced by Module 2.

For example, for the input sentence that reads "read the Medication Guide *before* you start taking Ibuprofen", we recognize the following two predicates, call them B and C:

- B = read the Medication Guide.
- C = you start taking Ibuprofen.

These two predicates in the sentence are connected by the function word *before*. Writing this as a LingSem relation yields: *before*(C, B). As shown in Table 4.4, the function word *before* represents the process template of *Precedence*. Thus, this relation can be expressed as Precedence(B, C). However, the LingSem representation is less lossy as explained in Section 4.4.2.

It is important to note the following: even though LingSem is a less lossy semantic representation, that also makes it less usable for formal reasoning and machine processing in general. Keep in mind that this is an intermediate representation made for a specific purpose of human-readability. The framework aims to convert this further into a more machine-readable format.

With that context, we explain the two tasks in more detail below. The first task is to construct the predicates from the dependency tree. The second task is to draw out the linguistic operators/connectors from the dependency matches, determine their operands, and weave the nested hierarchy that connects multiple linguistic operators within one sentence—weave them into a tree of LingSem relations, i.e., a LingSem tree.

4.4.1 Task 1: Predicate Construction

Goal: To combine some sets of tokens in a dependency tree into predicates and drop the dependency relations between the combined tokens.

Input: A dependency tree and a set of pattern matches.

Output: A predicate tree; essentially a dependency tree consisting only of matched relations.

The first task of Module 4 is to traverse the dependency tree and construct all predicates of interest. The predicates of interest are those that are extracted from a matched dependency subtree. To elaborate, consider tokens that have a common head/ancestor in the dependency tree such that the ancestor is part of a match, but the tokens are not part of any match. To illustrate briefly from the running example, Figure 4.10 shows the phrase 'each time' as part of a match but its descendant tokens 'you get a refill' are not part of any match. Such tokens are grouped to form a predicate. A constructed predicate acts, then, as an operand of the LingSem relation which the match translates to.

The procedure of this task is captured in Algorithm 1 and explained as follows. The first step (Lines 1–3) iterates through the matches and a set of all matched tokens T_m is constructed. Each matched token is a node in the dependency tree, we call a matched node. Next, in the second step (Lines 4–6), the subtree of each matched node is computed and stored in a set called *Subtrees*. The third step (Lines 7–10) is pruning the subtrees. For each subtree r (whose head is a matched node n), all other subtrees are subtracted from r. What remains of r after the subtractions (if any) is the pruned subtree of of n. The fourth step (Lines 11–12) converts pruned subtrees to strings which represent predicates. Step 4 also produces the final output of the procedure which is a map P mapping every matched node to a candidate predicate. Table 4.5 shows the matched nodes and their corresponding predicates for our running example, explained below.

Regarding Line 8 of Algorithm 1, note that a node cannot have more than one parent in the tree. Therefore. for any two subtrees r, q, it is one of two cases: (1) one of the two subtrees is entirely contained in the other, i.e., r is a subset of q or vice versa; or (2) they do not intersect at all.

Performing Task 1 on the Running Example

The following explains how Task 1, predicate construction, is performed on the running example Sentence $\mathcal{S}_2$. The complete list of predicates we wish to extract for our running example is the following (B and C were mentioned above):

- A = your doctor has prescribed this medication.
- B = read the Medication Guide.
- C = you start taking Ibuprofen.
- D = each time that you get a refill.

To visualize how predicates are extracted from the pattern matches, observe Figure 4.10 which shows the tokens matched and highlights the three pattern matches. Consider the leftmost match. That is a match against the pattern called `advcl-if`. For each token in the match, the subtree is retrieved. Then, the subtrees of all other matched tokens are subtracted from it. Take the token *read* for example. Note that, since *read* is the head of the dependency tree, its subtree is the entire dependency tree. We subtract from it the subtrees of all other matched tokens, i.e., the subtree

Algorithm 1 Predicate construction algorithm.

Input: A sentence S, its dependency tree D_S and a set of matches M_s.
Output: A map P containing the predicate of every matched token.

▷ Step 1: Construct T_m, the set of tokens matched:
for each match m in M_s **do**
    if $t \in$ GetTokens(m) **then**
        $T_m \leftarrow T_m \cup \{t\}$

▷ Step 2: Construct *Subtrees*, the set of subtrees of matched tokens:
for each token $t \in S$ **do**
    if $t \in T_m$ **then**
        $Subtrees \leftarrow Subtrees \cup$ GetSubtree(t, D_S)

▷ Step 3: Prune the subtrees:
for each pair of trees (q, r) in *Subtrees* **do**
    if $r \subset q$ **then**
        $q_x \leftarrow q \setminus r$
        $Subtrees_{pruned} \leftarrow Subtrees_{pruned} \cup \{q_x\}$

▷ Step 4: Convert pruned subtrees to strings, making the desired output map P:
for each subtree r in $Subtrees_{pruned}$ **do**
    $P \leftarrow P \cup ($Head$(r) \mapsto$ TreeToString$(r, S))$

▷ Auxiliary Functions:
function GetSubtree(t, D) ▷ Given a token t and a dependency tree D such that t is a node in D, return the subtree of t.
function GetTokens(m) ▷ Given a match m, return tokens in m.
function Head(r) ▷ Given a tree r, return its head.

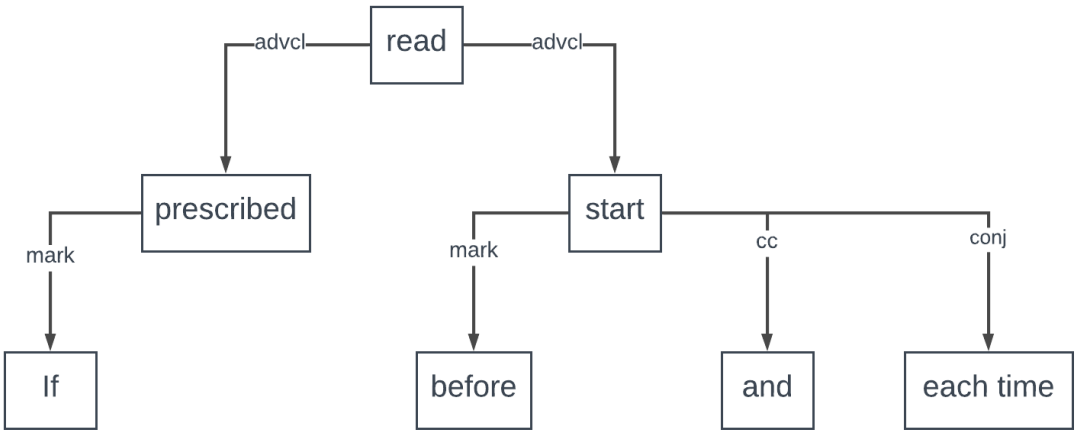

Figure 4.9: Tokens from the dependency tree of Figure 4.2 that are matched against dependency patterns.

Token matched	Predicate extraction	Result
read	$subtree(read) - subtree(prescribed) - subtree(start)$	read the Medication Guide.
prescribed	$subtree(prescribed) - subtree(if)$	your doctor has prescribed this medication.
if	$subtree(if)$	if.
start	$subtree(start) - subtree(before) - subtree(and) - subtree(each\ time)$	you start taking Ibuprofen.
before	$subtree(before)$	before.
and	$subtree(and)$	and.
each time	$subtree(each\ time)$	each time that you get a refill.

Table 4.5: Matched tokens from Figure 4.10, their predicates and the subtractions performed on each to achieve the predicate.

of *prescribed*, and the subtree of *start*. Once subtracted, we are left with the subtree $B =$ "read the Medication Guide".

Apply the same principle on the rest of the matched tokens. The predicate extracted from the token *prescribed* is its own subtree minus the token *if*. This and the rest are given in Table 4.5.

Sample Output of the Predicate Constructor

The following listing shows sample output of the matcher from Model 3 after it has been augmented with the predicate constructor.

```
Sentence 2: If your doctor has prescribed this medication, read the Guide before you start
    taking ibuprofen and each time that you get a refill.
   Match 1, Rule 4 (adv-clause 'before' pattern):
    main_clause_verb: read the Guide
```

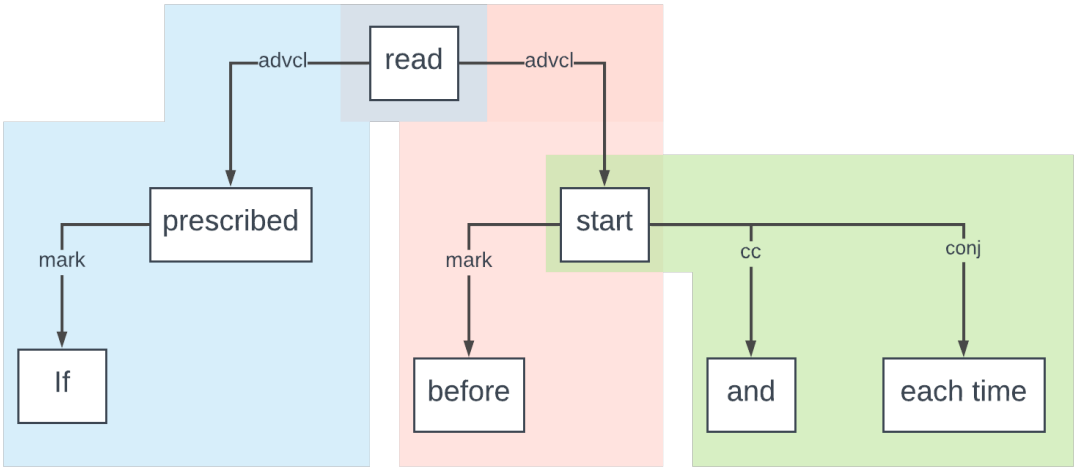

Figure 4.10: Tokens from the dependency tree of Figure 4.2 that are matched against dependency patterns, with the three matches highlighted.

```
    adv_clause_verb: you start taking ibuprofen and each time that you get a refill
    adv_clause_mark: before
  Match 2, Rule 7 (adv-clause 'if' pattern):
    main_clause_verb: read the Guide
    adv_clause_verb: your doctor has prescribed this medication
    adv_clause_mark: If
  Match 3, Rule 8 (and-conj-pattern):
    clause_1_verb: before you start taking ibuprofen
    clause_2_verb: each time that you get a refill
    conj_particle: and
```

Note the difference between this set of predicates (apparent in the output above and in Table 4.5) and the final predicates we seek (labeled A to D above). The difference is that each function word (*if*, *before*, *and*) has a predicate too, albeit the word itself. These are not considered predicates in the final output of Module 4 due to the role of Task 2. The reason is that Task 2 identifies these as connectors of predicates. Task 2 is explained in the next section.

4.4.2 Task 2: Building a LingSem Tree

Goal: To transform matches, which are now completed with predicates, into subtrees of LingSem relations, and arrange those in a semantic hierarchy based on the given dependency hierarchy.

Input: A predicate tree, a map from dependency pattern to LingSem relation, and a set of rules dictating precedence among LingSem relations.

Output: A LingSem tree; a tree of linguistic-semantics (LingSem) relations.

Example output: The LingSem tree shown in Figure 4.13

The second task of Module 4 is to construct a LingSem tree given a dependency tree and a set

of matches. A dependency tree naturally carries a nested hierarchy of lexical units (words) that is based on syntactic relations, whereas a LingSem tree's hierarchy is based on semantic relations that are derived from *the meanings* of said lexical units. At the same time, a LingSem tree preserves the lexical units as opposed to a Process Template for instance. This task is, in essence, about converting the syntactic dependency hierarchy into a semantic hierarchy. Take the following sentence for example:

> If you read and learn, you become smarter.

The dependency tree of this sentence is shown in Figure 4.11. Figure 4.12 shows the same tree with the pattern matches highlighted. The LingSem tree we aim to arrive at is shown Figure 4.13.

Design Decision: Making the Intermediate Representation of LingSem

The output of Module 3, the matched parts of a dependency tree, could be directly translated into the desired final formal outputs. That is done through writing a separate translator for each of the desired output languages, a translator that inputs dependency matches and outputs the formal expression. However, we chose to insert an intermediate representation, the LingSem tree, in between the dependency matches and the final formal outputs. This was motivated by multiple factors; briefly, it provides an intermediate representation which dependency trees are easy to translate to in a lossless way, and is itself easy to translate further into formal outputs. We explain these factors in more detail in the following.

Consider the following two example sentences.

> Sit down before you drink the medicine.

> Drink the medicine after you sit down.

Each of the two sentences contains the same two activities, *sitting* and *drinking*, in addition to one relation between the two activities. Both sentences would translate to the same Process Template, i.e., *Precedence*(sitting, drinking). However, through this transformation, much information has been lost—for starters, the conjunction particles *before* and *after*. The Process Template we crafted does not indicate which linguistic particle—*before*, *after* or another particle—is the one that carried the relevant semantics. In other words, the transformation is lossy and not reversible.

For this reason, we thought of creating an intermediate representation that is lossless, easy to read for both humans and computers, can be built from a dependency tree, and is easy to transform further into LTL and other formalisms. We created the LingSem tree displayed in Section 4.4 (Module 4) to fulfill these needs.

All in all, the addition of the LingSem tree format makes the design more modular and more extensible.

It is important to note that the used LingSem format is not a well-defined framework at the moment of writing. We discuss the future work of defining it in Section 6.3.

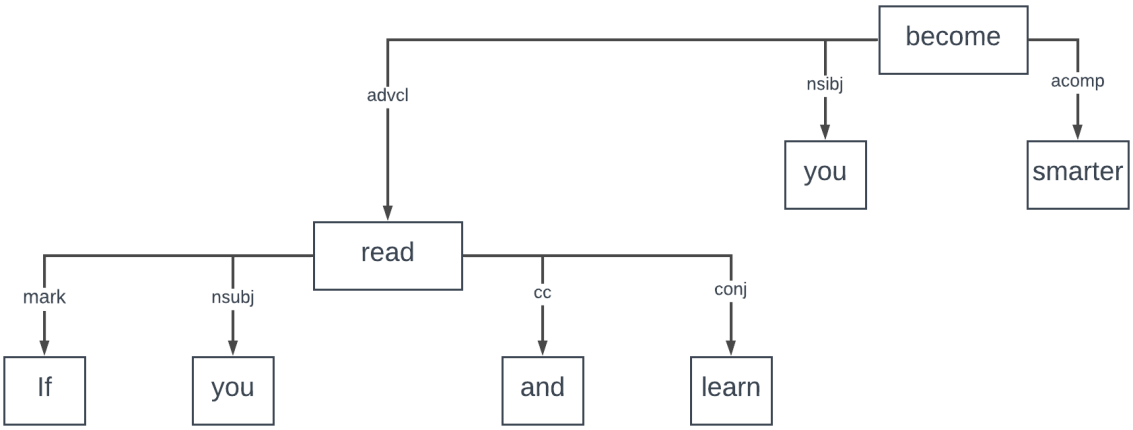

Figure 4.11: Dependency tree of the sentence “If you read and learn, you become smarter”.

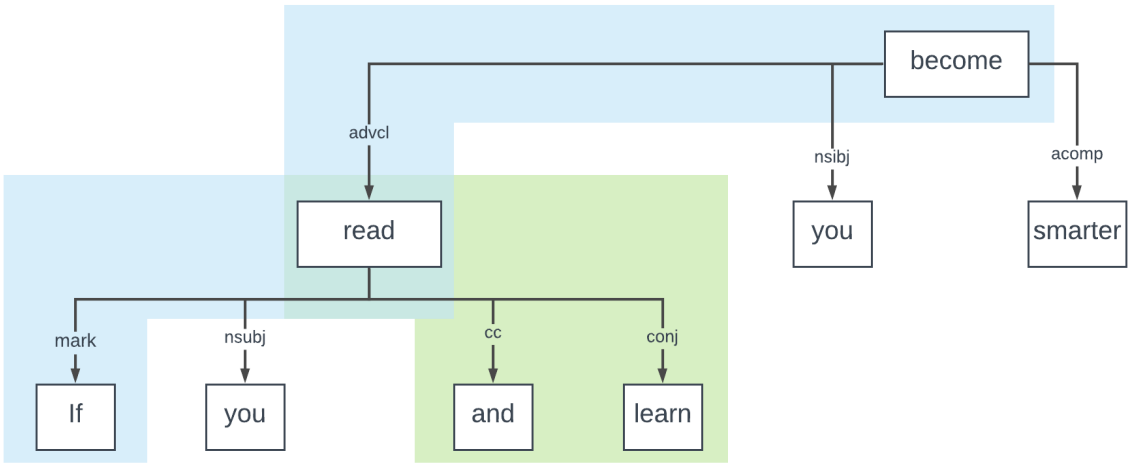

Figure 4.12: Dependency tree from Figure 4.11 with pattern matches marked.

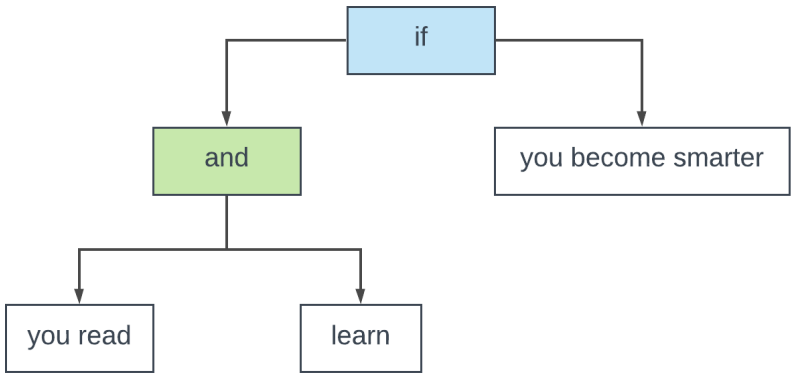

Figure 4.13: The LingSem tree constructed from the dependency tree and matches shown in Figure 4.12.

Precedence among Linguistic Patterns

As explained, we rely on the hierarchy of linguistic elements in a dependency tree in order to dictate the hierarchy of the LingSem tree. In some cases, however, different linguistic elements are on the same depth in the dependency tree. For instance, our running example tree (Figure 4.4) shows the *if*-clause on the same depth as the *before*-clause, and the two are siblings. In such cases, it is not clear which clause is nested within the other. To mitigate this, we introduce rules that govern which linguistic patterns have precedence over others, hence the 'precedence rules' resource shown in Figure 4.1. The term precedence here should not be confused with the process template of Precedence(x,y). A more detailed explanation with a briefer example follows to demonstrate the importance of precedence rules. Consider the following sentence labeled $\mathcal{S}_s$:

$\mathcal{S}_s$: If you are smart, listen and think, before you talk.

Observe the dependency tree of this sentence in Figure 4.14. The dependency analyzer sees the *before*-clause, the *if*-clause and the *and* conjunction, all on the same level in their dependency on the root *listen*. All three constructs are direct children of *listen*. However, in a tree of LingSem relations, we know that only one of these three words carries the semantics which will be at the root of the tree. And each of these three LingSem relations carries two operands. Therefore, there has to be some nesting among these three functions. For this example, there are six possible nesting permutations. The one that is intuitive to a human reader is represented in the following bracketed-string:

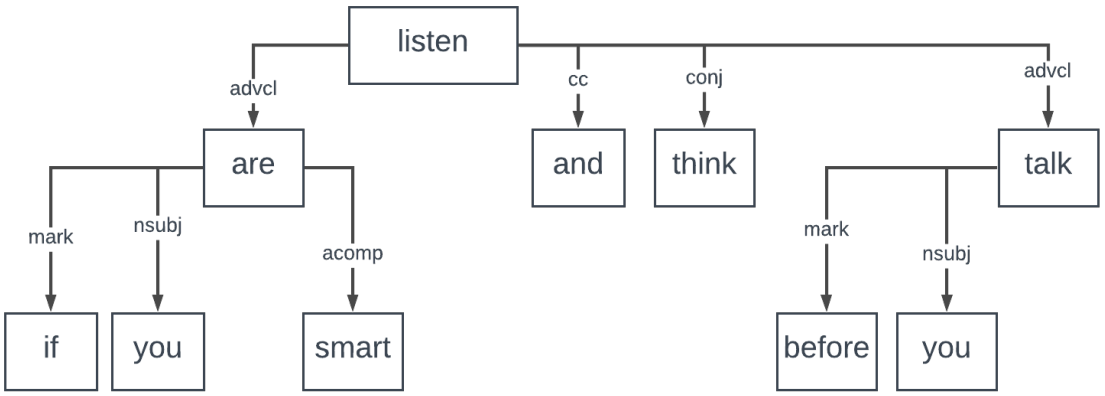

Figure 4.14: Dependency tree for sentence $\mathcal{S}_s$: "If you are smart, listen and think, before you talk"

if (you are smart, before (you talk, and (listen, think)))

However, in the absence of precedence rules, the machine's formalization of sentence $\mathcal{S}_s$ into a LingSem string may arbitrarily be any of the six permutations. In other cases with more dependency siblings, the number of permutations explodes exponentially. To mitigate this, we introduce precedence rules that govern how certain linguistic operators are nested within others.

Achieving the correct nesting of LingSem relations is essential. We demonstrate this through an example where the wrong nesting is used. Take the following permutation:

if (you are smart, and (before (you talk, listen), think))

This is interpreted as: "If you are smart, think, and before you talk, listen". This is not the intended meaning, however. That is the reason why deciding the right precedence is important.

Effect of precedence rules on running example.

To see the effect of precedence rules on our running example, observe the LingSem trees in the following two listings (also Figures 4.15 and 4.16) showing the LingSem tree output before and after the introduction of precedence rules respectively.

```
LingSem Tree:
Root: before.
|-- main-clause: if.
|   |-- if-clause: your doctor has prescribed this medication.
|   +-- then-clause: read the Guide.
+-- before-clause: and.
    |-- conjunct-1: you start taking ibuprofen.
    +-- conjunct-2: each time that you get a refill.

LingSem String:
before(and(C, D), if(A, B))
```

Listing 4.3: The LingSem tree of the running example before introducing precedence rules.

```
LingSem Tree:
Root: if.
|-- if-clause: your doctor has prescribed this medication,
+-- then-clause: before.
    |-- main-clause: read the Guide.
    +-- before-clause: and.
        |-- conjunct-1: you start taking ibuprofen.
        +-- conjunct-2: each time that you get a refill.

LingSem String:
if(A, before(and(C, D), B))
```

Listing 4.4: The LingSem tree of the running example after introducing precedence rules.

A limitation of the implemented mechanism of precedence rules

The rule introduced, which gives *before* higher precedence than *if*, does not always lead to the correct result. To illustrate the faultiness of the rule, consider the following two sentences:

- Think before you decide if you want to be wise.
- Think before you decide if you want to join.

These two sentences use different functions of the word *if*. The first is the marker of an adverbial-clause modifier. The second one is a complementizer and it works like *whether*. In the first

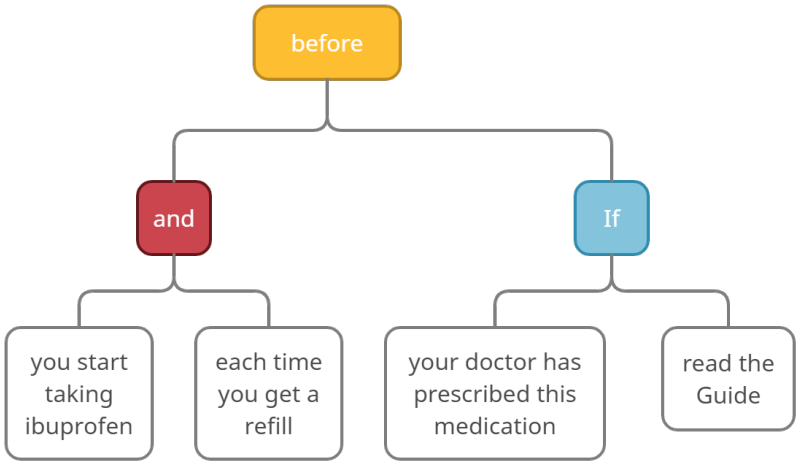

Figure 4.15: The LingSem tree of Sentence $\mathcal{S}_2$ before introducing the precedence rules.

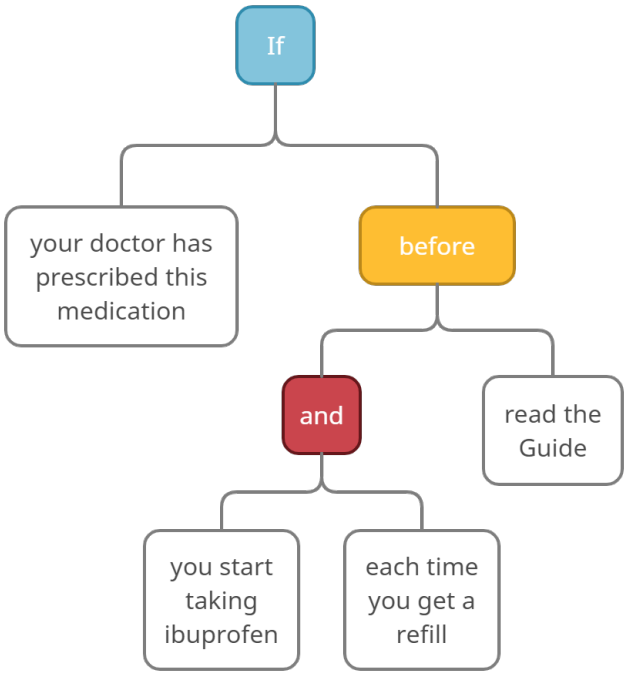

Figure 4.16: The LingSem tree of Sentence $\mathcal{S}_2$ after introducing the precedence rules.

example, the *before*-clause is nested within the *if*-clause, but in the second example, the opposite is true.

In light of that, we make clear that what we introduced is a simple prototypical mechanism that acts as a proxy to solving the problem of possible wrong nesting. We observed that the prototype works in some instances. However, it was not tested beyond a few examples as the above. We speculate that it needs more precise and more flexible rules. Additionally, the input of experts in linguistics and formal semantics is needed. Therefore, we leave this part for future work.

4.5 Module 5: Translator

Goal: To translate a LingSem tree into one or more formal languages; the current implementation supports one formal language, LTL. This module produces our final output, an LTL expression.

Input: A LingSem tree, and a set of dictionaries, each mapping linguistic LingSem relations to constructs from a certain formal language.

Output: A formal expression for each formal language supported.

Module 5 is the final module of our framework. It inputs the LingSem tree and outputs a final formal expression. The formal language supported for the small set of semantics we target is LTL. The module uses a dictionary as a resource for each output language. The LTL dictionary, for instance, maps each LingSem relation supported to an LTL expression. Recall that the nodes of a LingSem tree are either a LingSem relation or a linguistic predicate. The translator uses the dictionary to transform the LingSem relation nodes of the tree into the equivalent formal expression. After that, the tree is flattened through a breadth-first traversal, thus by generating the final formal output shown below. The tree can also be flattened before translating any node, which results in a *LingSem string*. The LingSem tree, the LingSem string and the LTL expression, all three outputs from the tool for Sentence $\mathcal{S}_2$ are shown in Listing 4.5 below.

```
LingSem Tree:
Root: if.
|-- if-clause: your doctor has prescribed this medication,
‘-- then-clause: before.
    |-- main-clause: read the Medication Guide provided by your pharmacist
    ‘-- before-clause: and.
        |-- conjunct-1: you start taking ibuprofen.
        ‘-- conjunct-2: each time that you get a refill.

LingSem String:
if(A, before(and(C, D), B))

LTL Formula:
G ( A -> F ( ~ (C & D ) W B ))
```

Listing 4.5: Final output of the tool for Sentence 2.

The use of dictionaries allows extending the translation to more formal languages as long as they are expressive enough for the sought semantics. This adds to the modularity of the framework. Moreover, the source notation of the dictionaries being a human-readable intermediate representation such as the LingSem tree adds to the extensibility of the framework.

The final formal outputs for all sentences of Table 5.1 are shown in the results in Chapter 5. The raw output of the tool is in Appendix C.

4.5.1 Determining the Desired Output Formalism

We have seen multiple options in related works in Chapter 3. We have seen LTL formulas, DECLARE relations, and even process model diagrams. We have chosen the former option for reasons explained below.

Design Decision: LTL Output

The following are reasons and motives for which LTL was chosen as a the primary output formalism:

1. It allows nesting.
2. It holds added value through its capacity for manipulation and rewriting as explained in Section 2.3.1.
3. It serves as a popular framework for formal reasoning and model checking [6].
4. The author's familiarity with LTL compared to other formalisms.
5. Time limitation; generating LTL requires less work than generating process model diagrams for instance.
6. LTL can extend to MTL (Section 2.3.1) by replacing four operators with time-constrained versions, which allows capturing real-time semantics [29]. This possibility of future work is detailed in Section 6.3.

For these reasons, we decided to produce LTL formulas as output.

4.5.2 The LTL Dictionary

We have a few sources of mappings from linguistic patterns or process templates (more the latter) to LTL expressions, Shafiee [53], Dwyer et al. [14], Maggi et al. [33], Pesic et al. [41] (See Figure 2.3), Bernardi et al. [5], and van der Aa [69]. Here we list our choices based on those sources and relying on our own judgement. We list the process templates and LTL expressions of interest, then we argue for a linguistic pattern that may carry the intended semantics.

It is important to note that the choices made do not cover the entire spectrum of meaning a linguistic pattern may carry. The LTL equivalents certainly do not capture every possible meaning of the corresponding linguistic patterns.

***Response*(A,B)**

The LingSem relation we chose for this process template is the conditional *if* construct, if(A,B). On the LTL side, according to Dwyer [12, 14], *Response*(A,B) translates to: $\mathcal{G}(A \rightarrow \mathcal{F}B)$.

***Precedence*(A,B)**

The linguistic pattern inferred from our understanding of the semantics of the *Precedence* process template is the one pertaining to the LingSem relation *before*(B,A). On the LTL side, the process template *Precedence*(A,B) is translated by [41] to the LTL expressions: $\neg B \ \mathcal{W} \ A$.

Additionally, We chose to treat the linguistic pattern *after*(A,B) as the opposite of *before*. Therefore, *after*(A,B) is equivalent to *before*(B,A). Tying *after*(A,B) to the process template of *Succession* was another choice, but it would yield a stronger (more constrained) LTL expression while the real semantics of *after* may be looser depending on natural language ambiguities.

The LTL Until operator

For the LTL expression: $A \ \mathcal{U} \ B$, the linguistic pattern inferred uses the *until* function word, making the corresponding LingSem relation *until*(B,A).

Other observed linguistic patterns

Some linguistic patterns were observed directly from the input. We chose to map their equivalent LingSem relations to LTL expressions as follows.

When(A,B): Although the function word *when* may carry different meanings as pointed out by Shafiee [53], we decided to choose the meaning of *whenever*. That is, always, B is true at the same time that A is true, without any implication on future or past states. That is the weakest LTL expression we found related to the natural meaning of when. Therefore, it is translated to: $\mathcal{G}(A \rightarrow B)$

Unless(B,A): According to [30], the linguistic pattern A *unless* B is expressed in LTL as: $A \ \wedge \ \mathcal{G}(B \rightarrow \mathcal{F} \ (\neg A))$.

Once(A,B): Once A is true, B is true now and always in the future. Therefore, we infer the LTL expression: $\mathcal{G}(A \rightarrow \mathcal{G}(B))$.

We only touch the surface with this selection of linguistic connectors. Much contribution is needed in this regard from the field of linguistics, particularly in identifying synonymy of different words. For example, it is useful to examine the synonymy of 'in case' to *if*, 'meanwhile' to *while*, 'except' to *unless*, and so on. Some pairs may be full synonyms (identical in every context), whereas others are partial synonyms (synonymous in some contexts) [32]. However, this thesis is concerned primarily with the principle of nested structures rather than coverage of connectors.

4.6 Resolution to Research Question

In this section, we investigate how the research question formulated in Section 1.2 is answered.

> **Research question:** How can nested process fragments be extracted from a sentence describing a procedure in natural language?

To achieve this, two main tasks were formed. To find what type of process fragments can be detected in a sentence and to find how to extract and formalize them.

First, the sentences get parsed by a dependency parser. Then we benefited from the Dwyer set of process templates (Section 2.3.3) to identify some processes templates (precedence, succession, response). Afterward, to discover the existing process fragments in the sentences, a list of possible conjunctions is collected. Then observing how these conjunctions have dependency relations to other tokens in a sentence, dependency patterns that correspond to each process template are formed. We match these dependency patterns against the input sentences using a dependency matcher. Out of that, we get a list of matches for each sentence. Then, we construct the predicates in a sentence by analyzing the matched and unmatched parts of the sentence. After that, we build a tree where each conjunction has its conjunct predicates as children, and where conjunctions are nested within one another following the hierarchy found in the dependency tree. We called this tree the linguistic-semantics (LingSem) tree. Then, we translated the conjunction subtrees (conjunction node with its predicate children) in the LingSem tree to the process templates decided and mapped earlier. We also generated LTL expression by translating each process template to its equivalent LTL expression. Through this pipeline, the nesting has propagated from the dependency tree to the LingSem tree to the final formal expressions.

To summarize, fragments of processes from sentences are extracted and formalized by defining different process templates and dependency patterns using a formal language and a list of linguistic conjunctions.

4.7 Limitations and Threats to Validity

In the following, we discuss possible limitations to our approach and threats to its validity.

4.7.1 Limiting our Observation to a Specific Domain

As mentioned in Section 4.2, automatically scanned dependency trees to help decide which dependency patterns we should adopt. Whether the observation is manual or automatic, there is a threat to validity in case the input belongs to a specific domain. In that case, certain elements may be more or less frequent than they would normally be in a typical procedural description text, and thus wrong assumption will have been made. In our case study, we do use a domain-specific input, which is not representative enough of procedural descriptions in general. This can be mitigated by applying our automatic observations using a systematic method and on a wide variety of domains. Nevertheless, most NLP projects target a certain domain [4, 6].

4.7.2 One Dependency Pattern Mapping to Multiple Meanings

Sometimes, one dependency pattern can indicate more than one linguistic meaning, which leads to different semantics. The problem is that if we cannot rely on the discrepancy of dependency relations to indicate a discrepancy in semantics, then we have to rely on human understanding

which is error-prone and slow. This problem is due to the ambiguity inherent in natural language. As far as we know, there is no way to mitigate this particular threat, except through using the most advanced dependency parser and the most accurate dependency parsing model available.

4.7.3 Relying Solely on Dependency Analysis

There is a threat that naturally extends from relying on dependency analysis alone. Dependency parsing models are statistical models. By definition, the most advanced parser cannot achieve a 100% accuracy in correctly relating tokens and correctly labeling the relations. The model used in this thesis boasts an accuracy for unlabelled dependency relations of 95% and labelled dependency relations of 94%. Thus, when the used dependency parser misplaces and/or mislabels a dependency relation (which is inevitable due to the statistical nature of the model), then the error propagates to the final output. This can be mitigated by employing other means of semantic analysis alongside dependency analysis.

4.7.4 The Limitation of Relying on Function Words Alone

It is especially evident in the the running example sentence that, in order to detect process-related semantics, it suffices to observe function words such as *if* and *before*. Consider, on the other hand, the following semantically-similar sentence.

> On the condition that your doctor has prescribed this medication, reading the Medication Guide must precede taking Ibuprofen.

The semantics of *Succession* are not obvious to a machine that does not understand the meaning of the word *precede*. Our framework with its current approach would be blind to the *Succession* semantics in this sentence. Although we would likely never face such a contrived sentence in the real world, we produced it merely to convey the point that a higher level of semantic analysis may be required to reveal process information that function-word matching cannot.

4.7.5 Relying on Human Interpretation

A threat to validity of our approach is that it relies completely on human understanding and interpretation in order to define the transformation rules. Specifically, the area of defining which linguistic patterns map to which LingSem relations. Ideally, this should rely on a sound theory mapping linguistic constructs to formal expressions. The obvious reason behind this threat is the lack of such theory. The state of the art uses approximation approaches, machine-trained models, and the like [4]. There remains a great deal of human intuition required to accurately transform a sentence into its formal equivalent [38]. The ambiguities induced by natural language shown in Section 2.1 demonstrates that.

4.8 Further Discussions

This section is designated to further discussions that were excluded from the main explanation of the approach for brevity.

4.8.1 The Accuracy of the Dependency Parser (Module 2)

The currently-used dependency parsing model has some limitations which we observed through experimentation. Briefly, in some cases, changing individual words to semantically equivalent ones produces different dependency trees which imply drastically different meanings. The following explores such a case in detail.

The tree in Figure 4.2 includes one wrong relation. To spot it, we need to analyze the tree using our understanding of language. We read a dependency tree starting from the root. The root here is the verb *read*. It has five children:

1. an adverbial-clause modifier (`advcl`), that is the verb *prescribed*,
2. a direct object (`dobj`), that is *the Medication Guide*,
3. an adverbial-clause modifier (`advcl`), that is the verb *start*,
4. a coordinating conjunction (`cc`), which is *and*,
5. and a conjunct (`conj`), the verb *get*.

When reading a dependency tree as a human, it is important to realize that a relation towards a token does not stop at that token, but rather extends to the entire subtree of that token. A *subtree* of a node in a dependency tree is composed of the node itself and all its descendants. For example, when we say that the verb *start* is an adverbial-clause modifier to the verb *read*, we realize that the modifier of the verb *read* is not only the verb *start*, but rather its entire subtree, i.e. the entire clause "before you start taking ibuprofen".

With that, we realize that the clause "each time you get a refill" is in conjunction with "read the Medication Guide". That means that, in effect, we should expect that the sentence, not modified by the two adverbial clauses, should read as: "read the Medication Guide and each time you get a refill". But that makes no sense to a human reader. Inspecting the subtree of the second clause further provides the answer. Notice that *each time* is a noun-phrase adverbial modifier (`npadvmod`) to the verb *get*. In essence, that means that *each time* modifies your *getting* of a refill. In that case, the proper way to read the sentence is: "read the Medication Guide and you get a refill each time" or "read the Medication Guide and, each time, you get a refill". The meaning conveyed in both cases is the same, yet it is the wrong meaning according to our understanding as human readers.

By studying this dependency tree, we now know that the machine understands this sentence closer to the following formulation:

> If your doctor has prescribed this medication, read the Medication Guide before you start taking Ibuprofen and you would get a refill each time you do that.

This is obviously not the intended meaning. To find out how the correct dependency tree should look like, we simplify the sentence and reduce its complexity and ambiguity to minimize the probability of error by the machine. Consider the following simplified variant:

> $\mathcal{S}_{2_1}$: If your doctor has prescribed this medication, read the Guide now and each time you get a refill.

Now observe the dependency tree of this sentence in Figure 4.3. We replaced the adverbial-clause modifier by a single word adverbial modifier (`advmod`), simply an adverb. We see two major changes. The first change is that the verb *read* is no longer in conjunction with the clause *each time you get a refill*. In fact it is not a conjunct at all, i.e., it is not related to any token through the `conj` relation. Instead, the adverb *now* is the one in conjunction with the aforementioned clause. The second major change is the dependency subtree of the aforementioned clause itself. Previously, the head of that subtree was the verb *get* whereas *each time* was considered an adverbial modifier to *get*, much like how *now* modifies *read* in this sentence. This time, however, the head of the subtree is the noun phrase *each time* with *each time* $\xrightarrow{\text{relcl}}$ *get*. This relation[4] indicates that *you get a refill* is a relative clause modifying *each time*. Effectively, the subtree carries a meaning more explicitly expressed as: "each time that you get a refill", which is the correct intended meaning of the original input.

To try and sway the machine into producing the correct dependency tree, we made a small modification to the input sentence, which we believe is disambiguating according to our investigation. We added the complementizer *that* such that the clause becomes "each time *that* you get a refill". The resulting sentence is as follows:

> $\mathcal{S}_{2_2}$: If your doctor has prescribed this medication, read the Medication Guide before you start taking Ibuprofen and each time that you get a refill.

The resulting dependency tree is shown in Figure 4.4. This tree carries the correct relations as described above, namely, *read* $\xrightarrow{\text{advcl}}$ *start* $\xrightarrow{\text{conj}}$ *each time* $\xrightarrow{\text{relcl}}$ *get*.

For the sake of completion, we experimented with a few more modifications which gave results as follows:

- Changing *each* to *every*: "*every* time you get a refill" produced a wrong tree.
- Inserting the complementizer *that* to the previous modification, making it "every time *that* you get a refill", produced the correct tree.
- Using the adverb *whenever* as "whenever you get a refill" produced a wrong tree.

[4]Note that `relcl` is called `rcmod` in the CLEAR style manual

Chapter 5

Results and Evaluation

In this chapter, the proposed method discussed in Chapter 4 is evaluated. To achieve this goal, we aim to perform a qualitative analysis by comparing the output of our tool with that of a human participant and investigating whether our method provides accurate formal expressions. In the following sections, the case study and analysis procedure are presented, the two sets of output are compared, the results are discussed, and the modifications on the implementation that followed are given.

5.1 Case Study Text

For evaluation, we chose a text of 360 words in 18 sentences describing 'how to use Ibuprofen oral'. The text is part of the WebMD online medical resource and can be viewed online[1]. The text is segmented into sentence and shown in Table 5.1.

A manual analysis of the text deemed it suitable as a case study for the following reasons:

- It qualifies as a procedural description text.
- Almost all sentences of the text included an ordering of activities or a decision, which are the process model semantics we aim to detect.
- Most of the sentences contained nested linguistic constructs.

Therefore, the text is considered suitable as input to our solution explained in Chapter 4.

[1] https://www.webmd.com/drugs/2/drug-5166-9368/ibuprofen-oral/ibuprofen-oral/details

$\mathcal{S}_1$	If you are taking the over-the-counter product, read all directions on the product package before taking this medication.
$\mathcal{S}_2$	If your doctor has prescribed this medication, read the Guide before you start taking Ibuprofen and each time you get a refill.
$\mathcal{S}_3$	If you have any questions, ask your doctor or pharmacist.
$\mathcal{S}_4$	Take this medication by mouth, usually every 4 to 6 hours with a full glass of water (8 ounces/240 milliliters) unless your doctor directs you otherwise.
$\mathcal{S}_5$	Do not lie down for at least 10 minutes after taking this drug.
$\mathcal{S}_6$	If you have stomach upset while taking this medication, take it with food, milk, or an antacid.
$\mathcal{S}_7$	The dosage is based on your medical condition and response to treatment.
$\mathcal{S}_8$	Take this medication at the lowest effective dose for the shortest possible time to reduce your risk of stomach bleeding and other side effects.
$\mathcal{S}_9$	Do not increase your dose or take this drug more often than directed by your doctor or the package label.
$\mathcal{S}_{10}$	For ongoing conditions such as arthritis, continue taking this medication as directed by your doctor.
$\mathcal{S}_{11}$	When ibuprofen is used by children, the dose is based on the child's weight.
$\mathcal{S}_{12}$	Read the package directions to find the proper dose for your child's weight.
$\mathcal{S}_{13}$	Consult the pharmacist or doctor if you have questions or if you need help choosing a nonprescription product.
$\mathcal{S}_{14}$	For certain conditions (such as arthritis), it may take up to two weeks of taking this drug regularly until you get the full benefit.
$\mathcal{S}_{15}$	If you are taking this drug "as needed" (not on a regular schedule), remember that pain medications work best if they are used as the first signs of pain occur.
$\mathcal{S}_{16}$	If you wait until the pain has worsened, the medication may not work as well.
$\mathcal{S}_{17}$	If your condition persists or worsens, or if you think you may have a serious medical problem, get medical help right away.
$\mathcal{S}_{18}$	If you are using the nonprescription product to treat yourself or a child for fever or pain, consult the doctor right away if fever worsens or lasts more than 3 days, or if pain worsens or lasts more than 10 days.

Table 5.1: All sentences comprising the sample input text of Ibuprofen usage.

5.2 Result Analysis Procedure

In the following section, we make a qualitative analysis on the output of our tool comparing it with the output of the human participant. We choose to make the analysis qualitative because of the many factors involved in transforming natural language to an LTL output. First and foremost, there is human understanding and knowledge of context which the machine does not possess. Second, in some cases there is more than one correct answer. Third, there are multiple steps in the translation process which can produce errors. A human may misidentify predicates, misinterpret the meaning, or be caught in multiple levels of nesting. On the other hand, the machine may err in the dependency parse, in generating predicates, nesting them correctly, and most importantly, interpreting the linguistic elements. Therefore, we found it more fruitful to analyze the results quantitatively in depth in light of these steps and these points of failure.

We gave the case study text to our human participant and asked for a translation to the intermediate LingSem format and to LTL formulas. For each sentence, the participant defined predicates, looked for semantics that could be expressed in LTL, and expressed them in LingSem format and the equivalent LTL expression based on the best of her judgement.

5.3 Results

This section shows certain input sentences and the corresponding outputs by our tool as well as by the human participant. The output for each sentence is followed by a discussion assessing the correctness of the output. In case of a mistake, we provide corrections and clarify which modules contributed to the mistake. To refer to output LTL formulas, we use the notation f_{nh} for human output of sentence $\mathcal{S}_n$; and f_{nm} for machine output of sentence $\mathcal{S}_n$.

The results chosen for display in this section are the ones perceived to be most insightful for the context of this thesis. Other results are deferred to Appendix A in order to avoid repetitive-, or otherwise less interesting, outcome.

5.3.1 Sentence $\mathcal{S}_1$:

"If you are taking the over-the-counter product, read all directions on the product package before taking this medication".

Predicates: The predicates extracted by both human and machine match.

A = you are taking the over-the-counter product
B = read all directions on the product package
C = taking this medication

Formal Expressions:

Format	Human output	Machine output
LingSem	*If* $(A,$ *before* $(C, B))$	*If* $(A,$ *before* $(C, B))$
LTL	$\mathcal{G}(A \rightarrow \mathcal{F}(B \rightarrow C))$	$\mathcal{G}(A \rightarrow \ \mathcal{F}\ ((\neg\ C)\ \mathcal{W}\ B))$

Discussion: We see a discrepancy between the two LTL outputs. Upon some analysis, we find that the human LTL output, formula f_{1h}, accepts traces such as $t_1^1 = \{A, \neg B\} \rightarrowtail \{\neg B, C\}$. This trace allows, given the precondition of A, that C occurs without B, i.e., taking the medication without reading the directions. This directly contradicts the sentence.

We also find that f_{1h} does not accept traces as $t_2^1 = \{A, B, \neg C\} \rightarrowtail \{B, \neg C\}$ where the run terminates with B having occurred but C never occurring. This contradicts the sentence which allows reading the directions without ever taking the medication.

Let us assume that the human made a mistake and meant to write the LTL formula $f'_{1h} = \mathcal{G}(A \rightarrow \mathcal{F}(C \rightarrow B))$. In this case, we face another problem; f'_{1h} does not accept traces as $t_3^1 = \{A\} \rightarrowtail \{B, \neg C\} \rightarrowtail \{\neg B, C\}$. In this trace, given A, C is allowed to occur some time after B without them occurring at the same time. This should be accepted by the LTL formula because the sentence allows it. This mistake can be blamed on the human's LTL translation of the LingSem relation *before* being wrong.

Regarding the machine's output, all the above mentioned traces t_1^1, t_2^1 and t_3^1 do not pose a problem. This is due to the fact that our translation of *before* in the LingSem-to-LTL dictionary (Section 4.5) is based on the assumption that *before* captures the semantics of Precedence; meanwhile, in $\mathcal{S}_1$, the function word *before* precisely captures the semantics of Precedence.

The dictionary of Module 5 (Section 4.5.2) dictates that the LingSem relation *if*(ψ, ϕ) translates to the LTL expression $\mathcal{G}(\psi \rightarrow \mathcal{F}(\phi))$. This allows traces such as $\{\psi\} \rightarrowtail \{\neg\phi\} \rightarrowtail \{\phi\}$, i.e., traces where the consequent of the *if*-statement is not effective immediately, but starts taking effect at a later state. This is due to the use of the *eventually* ($\mathcal{F}$) operator. Applying this concept to f_{1m}, we find that it accepts traces as $t_4^1 = \{A, \neg B, \neg C\} \rightarrowtail \{\neg B, C\} \rightarrowtail \{B, \neg C\}$ where the precedence of B over C is not required as soon as A holds but eventually after A holds. A stronger alternative translation of *if*(ψ, ϕ) is more suitable for this sentence, namely the LTL formula $\mathcal{G}(\psi \rightarrow \phi)$. With this modification, the LTL translation for $\mathcal{S}_2$ becomes:

$$f'_{1m} = \mathcal{G}(A \rightarrow (\neg C \ \mathcal{W} \ B))$$

We would like to adopt this correction into our framework such that it is performed systematically. This challenging problem is discussed in the context of the following sentence $\mathcal{S}_2$.

5.3.2 Sentence $\mathcal{S}_2$:

"If your doctor has prescribed this medication, read the Medication Guide provided by your pharmacist before you start taking ibuprofen and each time that you get a refill".

Predicates: The predicates extracted by both human and machine match.

A = your doctor has prescribed this medication
B = read the Medication Guide provided by your pharmacist
C = you start taking ibuprofen
D = each time that you get a refill.

Formal Expressions:

Format	Human output	Machine output
LingSem	*If* $(A, \textit{before}\ (\textit{and}\ (C, D), B))$	*If* $(A, \textit{before}\ (\textit{and}\ (C, D), B))$
LTL	$\mathcal{G}(A \rightarrow \mathcal{F}(B \rightarrow (C \vee D)))$	$\mathcal{G}(A \rightarrow\ \mathcal{F}\ (\neg\ (C \wedge D)\ \mathcal{W}\ B))$

Discussion: First we note that the LingSem relations produced by human and machine match. Upon initial inspection, we judge that the LingSem output is correct. This is the cumulative work of Modules 1 to 4 as shown in detail throughout Chapter 4 since it uses this sentence $\mathcal{S}_2$ as a running example. The output of Module 5, the LTL formula f_{2m}, differs from that of the human participant, f_{2h}. The human translated the *before* relation exactly as she did in Sentence $\mathcal{S}_1$. So did the machine. However, on translating the *and*(C, D) LingSem relation, she chose the logical *or* as the conjunction between C and D whereas the machine used the logical *and* directly mapped from the linguistic *and*.

Both LTL expressions are wrong because they allow traces $t_1^2 = \{A, \neg B\} \rightarrowtail \{\neg B, C\}$ and $t_2^2 = \{A, \neg B\} \rightarrowtail \{\neg B, D\}$ which clearly contradict the sentence requiring B to precede either of C or D. The mistake in f_{2h} is caused by the wrong translation of *before*(A, B) into the logical expression $A \rightarrow B$ which is discussed in Sentence $\mathcal{S}_1$ above. However, following the human in her use of the logical *or* operator can repair the machine's formula. The corrected formula is:

$$f'_{2m} = \mathcal{G}(A \rightarrow\ \mathcal{F}\ (\neg\ (C \vee D)\ \mathcal{W}\ B))$$

With this change, the traces t_1^2 and t_2^2 are no longer counterexamples because f'_{2m} does not accept them.

Another mistake remains that is the translation of the *if* relation which is discussed in $\mathcal{S}_1$. As does formula f_{1m}, formula f'_{2m} allows traces where the consequent of the *if*-statement is not effective immediately, but starts taking effect at a later state. Thus, f'_{2m} accepts traces as $t_3^2 = \{A, \neg B\} \rightarrowtail \{\neg B, C\} \rightarrowtail \{B, C\}$. Again, this allows starting to take the medication before reading the Medication Guide. To repair this mistake, we simply remove the *eventually* ($\mathcal{F}$) operator, making the expression as follows:

$$f''_{2m} = \mathcal{G}(A \rightarrow\ (\neg\ (C \vee D)\ \mathcal{W}\ B))$$

So far, we have performed two corrections to f_{2m}, one regarding the translation of *and*, and another regarding the translation of *if*. Naturally, we would like to adopt such corrections into our framework such that they are performed systematically. In the following, we discuss the possible solutions.

Solutions regarding translation of *and*: Regarding the wrong translation of *and*, we discuss two possible solutions. The first solution works at Module 5 and aims to deduce the correct logical conjunction from the linguistic one. Particularly, the aim is to map each instance of the linguistic conjunctions *and* and *or* into one of three logical operators: $\wedge, \vee, \nleftrightarrow$, where $\nleftrightarrow$ is the exclusive disjunction (Exclusive OR, XOR). This task is deemed too difficult, by Ghazel et al. [21], to solve

automatically. In their work, they relied on human input to solve the task. The reason is that deducing the logical meaning of the conjunction words *and* and *or* needs knowledge of context [50]. The conjunction *and* is ambiguous because, in many cases, it is not clear which words, phrases or clauses are the conjuncts [44]. Indeed, the wrong dependency tree (Figure 4.2) originally produced by Module 2 for this sentence was due to the ambiguity of the conjunction *and* (discussed at length in Section 4.8.1).

This sentence ($\mathcal{S}_2$) poses a rather complex case because of the implicit role of the conjunction *and* which is to list the times where predicate B, i.e. reading the Medication Guide, must hold. This semantic role of *and* can be simply demonstrated in the sentence:

> $\mathcal{S}_{2a}$: Read at dawn and at sunset.

Here, the reading is done if either of the two events, *dawn* or *sunset*, holds. Therefore, we translate it to the logical expression $(Dawn \vee Sunset) \rightarrow Read$. Notice that the word *and* translates to the logical operator $\vee$. Notice also that the sentence carries and implicit conditional statement that translates to the logical operator $\rightarrow$.

It is not clear what factor exactly dictates that the word *and* in sentences $\mathcal{S}_2$ and $\mathcal{S}_{2a}$ should translate to the logical $\vee$. We speculate that it is due to the fact that the *and* conjunction in both sentences is an adverbial clause that expresses time or temporal events [63]. However, we could not find counterexamples, nor could we reach a conclusion based on this speculation. More input is needed to solve this problem from experts in linguistics and logic. To the best of our knowledge, solving such ambiguity is still an open problem [21, 28, 50]. Therefore, we leave this part for future work.

The second solution works at Module 4 and it aims to alter the LingSem tree such that the subsequent processing (Module 5) produces the desired output. The current LingSem output (*If* $(A,$ *before* $($*and* $(C, D), B)))$ is visualized in Figure 4.16. A meaningful alteration of the LingSem output is *If* $(A,$ *and* $($*before* $(C, B),$ *before* $(D, B)))$ which is visualized in Figure 5.1.

The LTL expression that Module 5 produces out of this LingSem tree is the following (considering we also applied the correction for the translation of *if*):

$$f'''_{2m} = \mathcal{G}(A \rightarrow \ ((\neg\ C\ \mathcal{W}\ B) \wedge (\neg\ D\ \mathcal{W}\ B)))$$

This formula f'''_{2m} is equivalent to f''_{2m} due to De Morgan's law and the distributivity of $\mathcal{W}$ over $\vee$. Thus, the alternative LingSem tree is correct. What remains then is the ability to systematically reproduce such alteration. Observe the difference between the original tree (Figure 4.16) and the altered one (Figure 5.1). Note that the alteration does not only involve placing *and* above *before* in the nesting, but it also fabricates a sibling *before* relation and duplicates the main clause of the sibling *before* relation. This was performed through human understanding and insight, one particular element of which is understanding the phrase 'each time' and understanding that predicate D is an event in conjunction through *and* with event C. We find that these realizations make the alteration hard to replicate systematically by the machine.

Solutions regarding translation of *if*: Removing the *eventually* ($\mathcal{F}$) operator is based on the perceived urgency of the consequent of an *if* statement. In $\mathcal{S}_2$, it is hard for the machine to

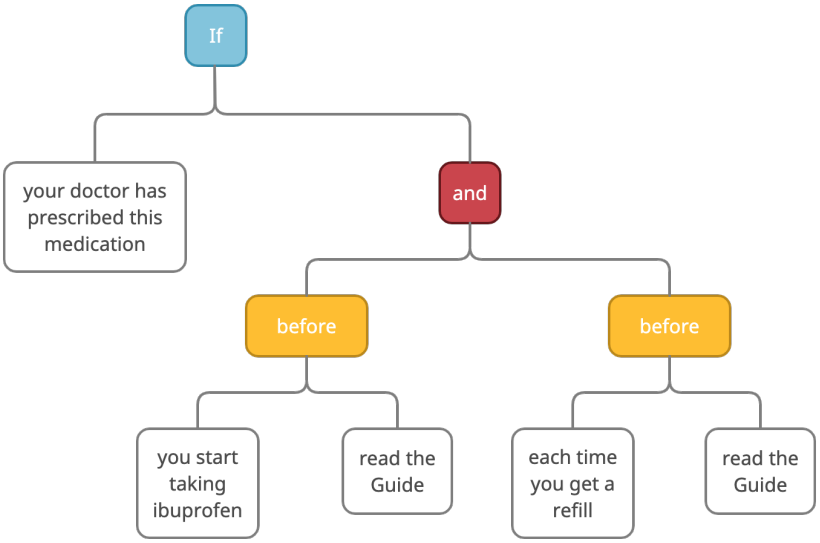

Figure 5.1: An alternative LingSem tree for $\mathcal{S}_2$.

perceive how important it is that predicate B, reading the guide, must precede predicates C and D. In another more relaxed sentence such as $\mathcal{S}_3$, the use of the *eventually* ($\mathcal{F}$) operator is the correct choice. Therefore, the choice of the correct LTL translation requires a level of semantic analysis deep enough to sense urgency. As this requires more research into semantic analysis, we leave it for future work.

We discussed possible solutions to mitigate the mistakes faced in processing $\mathcal{S}_2$. All of the discussed solutions require a deeper level of research and are thus part of future work.

5.3.3 Sentence $\mathcal{S}_{13}$:

"Consult the pharmacist or doctor if you have questions or if you need help choosing a nonprescription product".

Assumption: Here, the human participant decided to replace 'or if' by 'or' based on the assumption that the meaning would remain unchanged.

Predicates: The predicates extracted by both human and machine match, albeit with a small discrepancy for predicate C, which is labeled C_h and C_m for human and machine respectively.

A = Consult the pharmacist or doctor
B = you have questions
C_h = you need help choosing a nonprescription product
C_m = if you need help choosing a nonprescription product

Formal Expressions:

Format	Human output	Machine output
LingSem	*if* (*or* $(B, C), A$)	*if* (*or* (B, C), A)
LTL	$\mathcal{G}((B \vee C) \rightarrow \mathcal{F}(A))$	$\mathcal{G}((B \vee C) \rightarrow \mathcal{F}(A))$

Discussion (Redundant *if* word): Sentence $\mathcal{S}_{13}$ has two instances of *if*. The first one appears in the LingSem expression but the second does not. Call them the outer *if* and the inner *if* respectively. A complete *if* construct consists of two clauses, the *if*-clause and the main clause. In this sentence, the outer *if* construct was detected by Module 3 and can be seen in the LingSem expression. The *if*-clause of the outer *if* is "you have questions or if you need help choosing a nonprescription product" . It contains an *or*-conjunction of two clauses. This is also detected by Module 3 and can be seen in the LingSem expression. The second *or*-conjunct represents the *if*-clause of what we called the inner *if*. However, this *if*-clause has no corresponding main clause. So it does not represent a complete *if* construct. Indeed, looking at the dependency tree (Figure B.13), we find an `advcl` relation from the outer *if*-clause to the main clause, but we find no such relation from the inner *if*-clause. The same situation is found in Sentences 17 and 18. In these situations, the use of the word *if* in the inner clause is redundant. Thus, in this example, it suffices to say "if you have questions or you need help choosing...". This is why the human participant chose to ignore this redundant *if*. For the machine, however, Module 3 returned no match regarding the inner *if*. Consequently, Module 4 did not regard it while constructing the predicates. We do not consider this a mistake because of the redundancy of the inner *if* explained above.

The output of Modules 4 and 5 matches the human output and is correct.

Improvement: An improvement is possible here. As noted in the discussion of $\mathcal{S}_2$, conjuncts of an *or* may not be clear. The addition of the redundant inner *if* helps clear such ambiguity in this sentence. The sentence has one conditional statement marked by the outer *if*. The antecedent of that statement is "you have questions or you need help choosing a nonprescription product". The word *or* may interrupt the flow of reading the antecedent. To mitigate that, the writer repeats the word *if*. It will help the machine to recognize such deliberate disambiguation. However, by observing dependency trees of this sentence and others, we could not find a pattern identifying this type of disambiguation. Regardless, this falls in the line of research concerned with properly identifying conjuncts which relies on the dependency parsing model [31]. A dependency parsing model that is more advanced in detecting conjuncts would greatly benefit our framework for resolving such issues.

5.4 Discussion of Results

Here, we give some observations and explanations about the output shown in the previous section.

There are 4 sentences where the machine gave no output because it could not find meaningful patterns. Those are Sentences 7, 8, 10 and 12

Out of the 14 sentences where the machine found patterns and formed predicates, 12 agreed with the human participant. This counts the negation discrepancies as matching since the predicates

were matching before introducing the negation pattern. This is an indication of the success of dependency parsing and dependency pattern matching, Modules 2 and 3 respectively.

Regarding the LingSem output, considering the discrepancies in predicates formed, we can conclude that out of the 14 aforementioned sentences, the machine made no mistakes in finding the correct LingSem string. This indicates the success of Module 4.

Regarding the LTL output, many discrepancies appeared between human interpretation and the machine's simple dictionary mapping. Sentences 1, 2, 4, 5, 6, 11, 14, 17 and 18, all have nonequivalent LTL expressions for one reason or another. These differences were discussed under each respective sentence above. Only 5 LTL formulas agree between human and machine. Based on our inspection, we judged 6 out of 14 LTL formulas produced by the machine to have correctly conveyed the intended meaning of the sentence.

Chapter 6

Conclusion

In this chapter, we conclude our thesis by listing our contributions, discussing future work, summarizing the work, and giving concluding remarks.

6.1 Contributions

The contributions of this research are as follows:

1. The LingSem tree: An intermediate representation of natural language text that is both human-readable and machine-readable. That is a parse tree relating semantic units in a sentence. This representation is easier to translate than dependency trees into bracketed expressions of LingSem relations and temporal logics.
2. An extensible and modular framework consisting of five modules which utilizes dependency analysis to transform a natural language sentence into a LingSem tree, and from there on onto an LTL expression.
3. The framework is capable of detecting nesting in a sentence and transforming it into a nested formal expression.

6.2 Results

As of the time of writing this, the framework built through this thesis is capable of extracting nested process-fragments from a natural language sentence, and translating them into a nested LTL expression. It also outputs an intermediate readable format to aid in understanding and rectifying or extending the framework. The framework completes the objective in five modules. With a typical procedural description text as input, the first module uses a natural language processing (NLP) pipeline to pre-process the text, segment it into sentences, and annotate it with POS tags. The second module computes a syntactic dependency tree for each sentence. The third module detects, within the dependency tree, parts of interest which indicate process-related relations. For example, it detects clause markers, such as *if*, *whenever*, *before*, etc. and identifies

the heads of the clauses found in the sentence. This relies on an extensible set of rules defined externally in a JSON file. The fourth module uses a non-trivial algorithm to construct predicates from the matched heads of clauses and arrange them into a nested hierarchy of linguistic semantics relations, called a LingSem tree. That is our intermediate representation. The algorithm relies on a set of rules that determine the precedence among different linguistic elements. Finally, the fifth module utilizes extensible user-defined dictionaries to translate the LingSem tree into an LTL formula.

To make this clearer, we summarize the modules as follows:

- Module 1: Pre-processes the text, segments it into sentences and annotates it.
- Module 2: Extracts syntactic dependency information from the input text.
- Module 3: Detects subtrees of interest in the dependency tree according to pre-defined rules.
- Module 4: Constructs predicates and re-arranges them into a nested hierarchy, the LingSem tree.
- Module 5: Translates the LingSem tree to LTL given a dictionary; outputs the translated formulas.

The LingSem and the LTL outputs for 18 sample sentences with varying linguistic constructs were compared with those a human participant produced. Out of 18 sentences, the tool found dependency patterns of interest within 14 sentences (Modules 2 and 3). The LingSem output never mismatched with that of the human participant (Module 4) and was found correct in all cases. Finally, we judged the LTL output of 6 out of 14 sentences to be correct (Module 5).

6.3 Future Work

This section proposes ideas for future work.

6.3.1 Using Co-reference Resolution to Relate Multiple Sentences

Dependency relations are not derived *across* sentences. Dependency analysis is bound within a single sentence. That is mainly the source of the limitation that our analysis is bound within a sentence. However, we can still connect semantics across multiple sentences by using co-reference resolution. This is a technique that reveals which entities from a certain sentence are referred to in other sentences. This would allow us to relate the formal expressions of different sentences sharing the same reference, even though each expressions was derived from a single sentence.

6.3.2 Defining More Transformation Rules

In our attempt to build a complete framework that transforms process-related semantics to LTL expressions through multiple stages, the time constraint stood in the way of covering more semantics. As future work, more semantics need to be addressed, starting with more relations between process fragments followed by LTL semantics.

6.3.3 Probabilistic Precedence Rules

As pointed out in Section 4.4, probabilistic precedence rules are needed for a more accurate interpretation of implicit nesting of linguistic construct. Machine learning techniques, combined with a higher level of NLP, such as discourse analysis, can be used to train a such a model [3]. This is akin to how dependency grammars are trained.

6.3.4 Quality Control Measures

Misinterpreting natural language occurs even when using the latest technology in dependency analysis. Using other means of semantic and syntactic analysis alongside dependency analysis can act as quality control. When both methods differ in their interpretation, a flag can be raised for the input sentence to be manually revised. Another way to find misinterpreted sentences is to replace individual words with semantically similar ones, and observe if the dependency tree changes. The effectiveness of this approach is evident in Section 4.8.1 when replacing the phrase *each time* with *every time* or *whenever* yielded different dependency trees.

6.3.5 LingSem Definition

The LingSem tree format is not well-defined. Prior to defining it as a formalism, its requirements need to be explicitly stated. After that, similar semantic trees such as SemS [27] should be studied. In case an alternative can replace LingSem, it should be adopted. Otherwise, LingSem can adopt concepts from those similar but slightly different formalisms.

6.4 Concluding Remarks

This thesis has aimed to extract fragments of processes from natural language sentences and formalize them. It has succeeded in creating a modular and extensible framework relying on state-of-the-art techniques and tools to perform the task; yet it leaves much more to be desired. We hope to see more rules specified and advanced modules and techniques integrated into this skeleton of a framework.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. Data Structures and Algorithms. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1983. 9

[2] Catherine Anderson. Essentials of Linguistics. McMaster University, March 2018. 32

[3] Riza Theresa Batista-Navarro, Georgios Kontonatsios, Claudiu Mihăilă, Paul Thompson, Rafal Rak, Raheel Nawaz, Ioannis Korkontzelos, and Sophia Ananiadou. Facilitating the Analysis of Discourse Phenomena in an Interoperable NLP Platform. In International conference on intelligent text processing and computational linguistics, pages 559–571. Springer, 2013. 65

[4] Patrizio Bellan, Mauro Dragoni, and Chiara Ghidini. A Qualitative Analysis of the State of the Art in Process Extraction from Text. In AIxIA, pages 19–30, 2020. 16, 17, 48, 49

[5] Mario Luca Bernardi, Marta Cimitile, Chiara Di Francescomarino, and Fabrizio Maria Maggi. Using discriminative rule mining to discover declarative process models with non-atomic activities. In International Symposium on Rules and Rule Markup Languages for the Semantic Web, pages 281–295. Springer, 2014. 46

[6] Andrea Brunello, Angelo Montanari, and Mark Reynolds. Synthesis of LTL Formulas from Natural Language Texts: State of the Art and Research Directions. In Johann Gamper, Sophie Pinchinat, and Guido Sciavicco, editors, 26th International Symposium on Temporal Representation and Reasoning (TIME 2019), volume 147 of Leibniz International Proceedings in Informatics (LIPIcs), pages 17:1–17:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 9, 16, 46, 48

[7] Jinho D Choi and Martha Palmer. Guidelines for the clear style constituent to dependency conversion. Center for Computational Language and EducAtion Research, University of Colorado Boulder, Institute of Cognitive Science, Technical Report 01, 12, 2012. 19, 24

[8] P. Cserkuti, T. Levendovszky, and H. Charaf. Survey on Subtree Matching. In 2006 International Conference on Intelligent Engineering Systems, pages 216–221, London, UK, 2006. IEEE. 26

[9] Marie-Catherine De Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D Manning. Universal Stanford Dependencies: A cross-linguistic typology. In LREC, volume 14, pages 4585–4592, 2014. 19

[10] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008. 19

[11] Marc Denecker. On the informal semantics of knowledge representation languages and the case of Logic Programming. Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, 2020. 12

[12] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Proceedings of the second workshop on Formal methods in software practice - FMSP '98, pages 7–15, Clearwater Beach, Florida, United States, 1998. ACM Press. 11, 29, 47

[13] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in Property Specifications for Finite-State Verification. In Proceedings of the 21st international conference on Software engineering, pages 411–420, 1999. 11, 29

[14] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property Pattern Mappings for LTL. https://matthewbdwyer.github.io/psp/patterns/ltl.html. (Accessed on 08/14/2021). 11, 29, 46, 47

[15] IBM Cloud Education. What is Natural Language Processing? https://www.ibm.com/cloud/learn/natural-language-processing, July 2020. (Accessed on 08/19/2021). 7

[16] Elena Viorica Epure, Patricia Martín-Rodilla, Charlotte Hug, Rebecca Deneckère, and Camille Salinesi. Automatic process model discovery from textual methodologies. In 2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS), pages 19–30. IEEE, 2015. 14

[17] Michael Yoshitaka Erlewine. Elements of Formal Semantics: An Introduction to the Mathematical Theory of Meaning in Natural Language. Computational Linguistics, 42(4):837–839, December 2016. 9

[18] CK Español. What are the different levels of NLP? | Medium. https://medium.com/@CKEspanol/c0de6b9ebf61. (Accessed on 08/14/2021). 7

[19] Renato César Borges Ferreira, Lucinéia Heloisa Thom, and Marcelo Fantinato. A Semi-automatic Approach to Identify Business Process Elements in Natural Language Texts:. In Proceedings of the 19th International Conference on Enterprise Information Systems, pages 250–261, Porto, Portugal, 2017. SCITEPRESS - Science and Technology Publications. 15

[20] Fabian Friedrich, Jan Mendling, and Frank Puhlmann. Process model generation from natural language text. In International Conference on Advanced Information Systems Engineering, pages 482–496. Springer, 2011. 14

[21] Mohamed Ghazel, Jing Yang, and El-Miloudi El-Koursi. A pattern-based method for refining and formalizing informal specifications in critical control systems. Journal of Innovation in Digital Ecosystems, 2(1):32–44, December 2015. 18, 57, 58

[22] Aditya Ghose, George Koliadis, and Arthur Chueng. Process discovery from model and text artefacts. In 2007 IEEE Congress on Services (Services 2007), pages 167–174. IEEE, 2007. 13

[23] Joao Carlos de A.R. Goncalves, Flavia Maria Santoro, and Fernanda Araujo Baiao. Business process mining from group stories. In 2009 13th International Conference on Computer Supported Cooperative Work in Design, pages 161–166, April 2009. 13, 15

[24] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. Journal of the ACM, 29(1):68–95, January 1982. 26

[25] Krzysztof Honkisz, Krzysztof Kluza, and Piotr Wiśniewski. A Concept for Generating Business Process Models from Natural Language Description. In Weiru Liu, Fausto Giunchiglia, and Bo Yang, editors, Knowledge Science, Engineering and Management, volume 11061, pages 91–103. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science. viii, 15

[26] Dan Jurafsky and James H. Martin. Speech and Language Processing : an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Pearson Prentice Hall, Upper Saddle River, N.J., 2009. viii, 5, 8

[27] Sylvain Kahane and Nicolas Mazziotta. Dependency-based analyses for function words - Introducing the polygraphic approach. In Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015), pages 181–190, Uppsala, Sweden, August 2015. 65

[28] Matthew Knachel. Translating from English to Sentential Logic, 3 2021. [Online; accessed 2021-10-03]. 58

[29] Ron Koymans. Specifying real-time properties with metric temporal logic. Real-time systems, 2(4):255–299, 1990. 46

[30] Fred Kröger and Stephan Merz. Temporal logic and state systems. Texts in theoretical computer science : An EATCS series. Springer, Berlin, 2008. OCLC: ocn231678341. 10, 47

[31] Sandra Kübler, Ryan McDonald, and Joakim Nivre. Dependency parsing. Synthesis lectures on human language technologies, 1(1):1–127, 2009. 60

[32] John Lyons. Linguistic semantics: An introduction. Cambridge University Press, 1995. 47

[33] Fabrizio M. Maggi, Arjan J. Mooij, and Wil M.P. van der Aalst. User-guided discovery of declarative process models. In 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pages 192–199, Paris, France, April 2011. IEEE. 46

[34] Bilal Maqbool, Farooque Azam, Muhammad Waseem Anwar, Wasi Haider Butt, Jahan Zeb, Iqra Zafar, Aiman Khan Nazir, and Zuneera Umair. A Comprehensive Investigation of BPMN Models Generation from Textual Requirements—Techniques, Tools and Trends. In Kuinam J. Kim and Nakhoon Baek, editors, Information Science and Applications 2018, volume 514, pages 543–557. Springer Singapore, Singapore, 2019. Series Title: Lecture Notes in Electrical Engineering. 14

[35] Robert N Moll, Michael A Arbib, and Assaf J Kfoury. An introduction to formal language theory. Springer Science & Business Media, 2012. 9

[36] What is Natural Language Processing (NLP)? https://www.sas.com/nl_nl/insights/analytics/what-is-natural-language-processing-nlp.html. (Accessed on 08/19/2021). 5

[37] Mark Neumann. SpaCy's Dependency Matcher - An Introduction. http://markneumann.xyz/blog/dependency_matcher/. (Accessed on 10/06/2021). 27

[38] Nick Nicholas and John Woldemar Cowan. What Is Lojban? Logical Language Group, 2003. 49

[39] Joakim Nivre. Dependency grammar and dependency parsing. MSI report, 5133(1959):1–32, 2005. 8

[40] Arika Okrent. In the land of invented languages: Esperanto rock stars, Klingon poets, Loglan lovers, and the mad dreamers who tried to build a perfect language. Random House, 2009. 5

[41] Maja Pesic, Helen Schonenberg, and Wil van der Aalst. Declarative Workflow. In Arthur H. M. Hofstede, Wil M. P. Aalst, Michael Adams, and Nick Russell, editors, Modern Business Process Automation, pages 175–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. viii, 11, 12, 30, 46, 47

[42] Sampo Pietikäinen. Discovering Business Processes from Unstructured Text. Master's thesis, University of Jyväskylä, Jyväskylän, Finland, 2020. 15

[43] Amir Pnueli. The temporal semantics of concurrent programs. Theoretical Computer Science, 13(1):45–60, 1981. 9

[44] Maja Popović and Sheila Castilho. Are ambiguous conjunctions problematic for machine translation? In Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019), pages 959–966, 2019. 58

[45] Luis Quishpi, Josep Carmona, and Lluís Padró. Extracting Annotations from Textual Descriptions of Processes. In Dirk Fahland, Chiara Ghidini, Jörg Becker, and Marlon Dumas, editors, Business Process Management, volume 12168, pages 184–201, Seville, Spain, 2020. Springer International Publishing. Series Title: Lecture Notes in Computer Science. 17, 18, 19, 24

[46] Maximilian Riefer, Simon Felix Ternis, and Tom Thaler. Mining Process Models from Natural Language Text: A State-of-the-Art Analysis. Multikonferenz Wirtschaftsinformatik (MKWI-16), March, pages 9–21, 2016. 13, 15

[47] Kristin Y. Rozier. Linear Temporal Logic Symbolic Model Checking. Computer Science Review, 5(2):163–203, May 2011. 9, 10

[48] Amr Rekaby Salama, Özge Alaçam, and Wolfgang Menzel. Text completion using context-integrated dependency parsing. In Proceedings of The Third Workshop on Representation Learning for NLP, pages 41–49, 2018. 32

[49] Amr Rekaby Salama and Wolfgang Menzel. Learning context-integration in a dependency parser for natural language. In Intelligent Natural Language Processing: Trends and Applications, pages 545–569. Springer, 2018. 32

[50] Viola Schmitt. Boolean and non-boolean conjunction. The Wiley Blackwell Companion to Semantics, pages 1–32, 2020. 58

[51] Robert M Schwarcz. Towards a computational formalization of natural language semantics. In International Conference on Computational Linguistics COLING 1969: Preprint No. 29, 1969. iii, 1

[52] ScienceDirect. Formal Language - an overview | ScienceDirect Topics. https://www.sciencedirect.com/topics/computer-science/formal-language. (Accessed on 08/18/2021). 9

[53] Fatemeh Shafiee. Local Process Model Extraction from Textual Data. Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, September 2019. viii, 3, 9, 11, 14, 16, 30, 46, 47

[54] Avik Sinha and Amit Paradkar. Use cases to process specifications in business process modeling notation. In 2010 IEEE International Conference on Web Services, pages 473–480. IEEE, 2010. 14

[55] spaCy · Industrial-strength Natural Language Processing in Python. https://spacy.io/. (Accessed on 08/18/2021). 19

[56] spaCy Usage Documentation · Facts & Figures. https://spacy.io/usage/facts-figures. (Accessed on 08/18/2021). 19

[57] spaCy Models Documentation · English. https://spacy.io/models/en#en_core_web_trf. (Accessed on 08/18/2021). 24

[58] The Stanford Natural Language Processing Group – Stanford Dependencies. https://nlp.stanford.edu/software/stanford-dependencies.shtml. (Accessed on 08/19/2021). 19

[59] SemgrexPattern (Stanford JavaNLP API). https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/semgraph/semgrex/SemgrexPattern.html. (Accessed on 10/06/2021). 27

[60] Tgrep/Tgrep2 Tutorial. https://web.stanford.edu/dept/linguistics/corpora/cas-tut-tgrep.html#syntax. (Accessed on 10/06/2021). 27

[61] Josep Sànchez-Ferreres, Andrea Burattin, Josep Carmona, Marco Montali, and Lluís Padró. Formal Reasoning on Natural Language Descriptions of Processes. In Thomas Hildebrandt, Boudewijn F. van Dongen, Maximilian Röglinger, and Jan Mendling, editors, Business Process Management, volume 11675, pages 86–101. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science. 16, 17, 18

[62] Josep Sànchez-Ferreres, Andrea Burattin, Josep Carmona, Marco Montali, Lluís Padró, and Luís Quishpi. Unleashing textual descriptions of business processes. Software and Systems Modeling, May 2021. 18

[63] Sandra A Thompson, Robert E Longacre, and Shin Ja J Hwang. Adverbial clauses. Language typology and syntactic description, 2:171–234, 1985. 58

[64] Pallavi Tilloo, Raga Gottimukkala, and Sreeja Mamidala. Sentiment Analysis for Amazon Musical Instruments User Reviews. page 7, 2021. viii, 20

[65] Types of Gateway in BPMN. https://www.visual-paradigm.com/guide/bpmn/bpmn-gateway-types/. (Accessed on 09/01/2021). 17

[66] Universal Dependency Relations, UD version 2. https://universaldependencies.org/u/dep/. (Accessed on 08/19/2021). 19

[67] Universal POS tags. https://universaldependencies.org/u/pos/. (Accessed on 08/19/2021). 22

[68] Pashootan Vaezipoor, Andrew C. Li, Rodrigo Toro Icarte, and Sheila A. McIlraith. Ltl2action: Generalizing LTL instructions for multi-task RL. CoRR, abs/2102.06858, 2021. 2

[69] Han van der Aa, Claudio Di Ciccio, Henrik Leopold, and Hajo A. Reijers. Extracting Declarative Process Models from Natural Language. In Paolo Giorgini and Barbara Weber, editors, Advanced Information Systems Engineering, volume 11483, pages 365–382. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science. 13, 16, 46

[70] Wil Van Der Aalst. Data science in action. In Process mining, pages 3–23. Springer, 2016. 2, 9

[71] Wil van der Aalst. Process Mining: Data Science in Action. Springer Publishing Company, Incorporated, 2nd edition, 2016. 9

[72] W3Schools. JSON Structures | JSON Tutorial | W3Resource. https://www.w3resource.com/JSON/structures.php. (Accessed on 08/17/2021). 27

[73] Shira Wein and Paul Briggs. A Fully Automated Approach to Requirement Extraction from Design Documents. In 2021 IEEE Aerospace Conference (50100), pages 1–7, Big Sky, MT, USA, March 2021. IEEE. 18

[74] Wikipedia. Linear Temporal Logic. https://en.wikipedia.org/wiki/Linear_temporal_logic. (Accessed on 08/14/2021). 10

Appendix A

Results of Secondary Significance

This chapter shows the output of our tool for the remaining sentences of the input text—i.e., the ones not discussed in the body of the thesis (Chapter 5)—along with a discussion of the results.

A.1 Sentence $\mathcal{S}_3$:

"If you have any questions, ask your doctor or pharmacist".

Predicates: The predicates extracted by both human and machine match.

A = you have any questions
B = ask your doctor or pharmacist

Formal Expressions:

Format	Human output	Machine output
LingSem	$If\,(A, B)$	$If\,(A, B)$
LTL	$\mathcal{G}(A \rightarrow \mathcal{F}(B))$	$\mathcal{G}(A \rightarrow\ \mathcal{F}\ B)$

The machine outputs of Modules 4 and 5 are correct.

Undetected *and*/ *or* conjunctions. In Sentences 3, 6, 7, 8, and 9, the *and*/*or* conjunction was not detected because in all of these cases, the conjuncts are nouns while our pattern considers only verbs. This is intentional and the outcome we see here is desirable. The reason is that we intend to detect and link activities while activities are represented by verbs as explained in Section 4.3.2. The two conjunct nouns in $\mathcal{S}_3$ are objects of the activity 'ask'. Thus, they do not fall within the scope of our interest in this thesis.

A.2 Sentence $\mathcal{S}_4$:

"Take this medication by mouth, usually every 4 to 6 hours with a full glass of water (8 ounces/240 milliliters) unless your doctor directs you otherwise".

Predicates: The predicates extracted by both human and machine match.

A = Take this medication by mouth, usually every 4 to 6 hours with a full glass of water (8 ounces/240 milliliters)
B = your doctor directs you otherwise

In addition, the human participant defined the following predicate:

$\bar{A} = \neg A$ = Do not take this medication by mouth, usually every 4 to 6 hours with a full glass of water (8 ounces/240 milliliters)

Formal Expressions:

Format	Human output	Machine output
LingSem	*If* $(B, \neg A)$	*unless*(B, A)
LTL	$\mathcal{G}(B \rightarrow \mathcal{F}(\neg A))$	$A \wedge (\mathcal{G}(B \rightarrow \ \mathcal{F}\ (\neg A)))$

Both LTL outputs are wrong.

Discussion: The difference between the two LTL outputs is that the machine added the literal A in a logical *and* with the formula of the participant. This stronger expression is wrong simply because it does not allow the trace consisting of a single event B. The shortcoming stems from our particular mapping of *unless* in the LTL dictionary (Module 5).

The human's output is based on assuming *unless* to be synonymous with *if not*, and thus replacing the former with the latter. The resulting LTL expression is not correct because it allows the trace $\{B\} \rightarrowtail \{\neg A\} \rightarrowtail \{A\}$ which means that the patient is supposed to follow the doctor's direction for one time unit only. This contradicts our understanding of the sentence. To mitigate this, a possible formula is:

$$\mathcal{G}(B \rightarrow \mathcal{G}(\neg A))$$

However, this formula accepts the trace $\{\neg A\} \rightarrowtail \{\neg A\} \rightarrowtail \ldots \rightarrowtail \{\neg A\}$ where A, the initial instruction, never holds.

We argue that the *weak until* $\mathcal{W}$ operator would have been a more suitable translation for this sentence. Thus the correct formula is:

$$A \ \mathcal{W} \ B$$

It works in the following way. "Take the medicine" (A) is true and remains true until some time (not necessarily coming) when the doctor directs you otherwise (B); and when that (B) occurs, then A becomes false and remains false.

A.3 Sentence $\mathcal{S}_5$:

"Do not lie down for at least 10 minutes after taking this drug".

Predicates: The predicates extracted by both human and machine match.

A = Do not lie down for at least 10 minutes
B = taking this drug

Formal Expressions:

Format	Human output	Machine output
LingSem	*After* (B, A)	*After* (B, A)
LTL	$\mathcal{G}(B \rightarrow \mathcal{X}(A))$	$A \ \mathcal{W} \ B$

The human output is correct, but the machine LTL output is wrong.

Discussion: Here, the temporal constraint *10 minutes* should be expressed in the formal output, perhaps through Metric Temporal Logic, rather than be part of the predicate A. Presumably, the participant assumed that the next state is 10 minutes long, hence the *next* ($\mathcal{X}$) operator. The participant's LTL formula states that every time the drug is taken, in the next 10 minutes, one should not lie down. In that sense, we argue that this is an accurate translation.

On the other hands, the machine's output is wrong because it does not allow the trace $\{\neg A\} \rightarrowtail \{B\}$ while the sentence allows it. Not allowing this trace means not allowing lying down for 10 minutes until the drug is taken.

It may seem as if this misinterpretation stems from a wrong assumption we made while writing the LingSem to LTL dictionary; the assumption that the semantics of *after*(A, B) are equivalent to that of *before*(B, A) whereas it could simply mean *next* as the participant chose. However, the reason the machine got this wrong goes back to an earlier phase of the process, the dependency tree. Observe Figure A.1. Notice that the clause "after taking this drug" depends on the verb 'lie', thereby conveying the meaning "Do not lie down after taking this drug" independently of the time period specified by "at least 10 minutes". We argue that that the *after*-clause should entirely be part of the *for* prepositional clause such that it depends on the temporal determinant '10 minutes'. This arrangement can be seen in Figure A.2 which was produced by the dependency parser after switching around the main clause with the *after*-clause. That way, the open period specified by the *after*-clause is tightly bound by a fixed period of 10 minutes (or more).

Unfortunately, the machine analyzing the sentence after this modification does not yield a LingSem tree. The reason is twofold: first and foremost, the lack of a defined dependency pattern for the prepositional *for* clause and the lack of the necessary subsequent dependency-to-LingSem mapping. Second, since the *after*-clause lies inside the *for*-clause, nothing can be said in certainty about the inner construct without recognizing the semantics of the outer one.

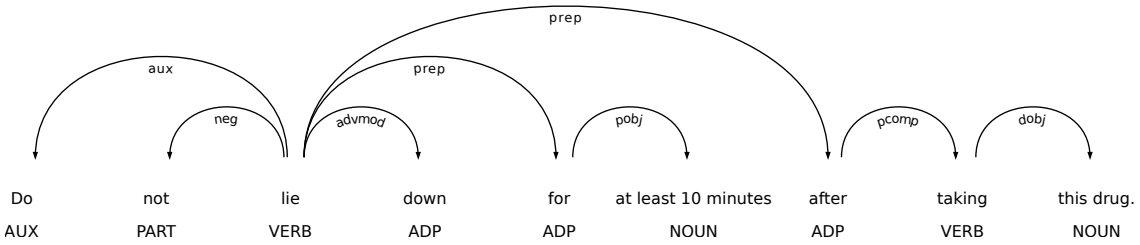

Figure A.1: Dependency tree of $\mathcal{S}_5$.

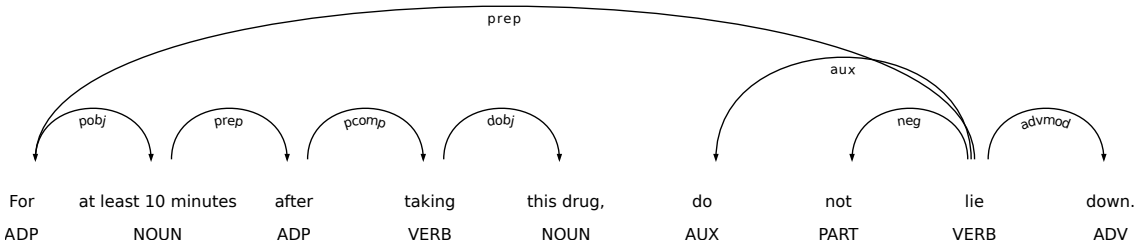

Figure A.2: Dependency tree of $\mathcal{S}_5$ after switching around the main clause with the *after*-clause.

A.4 Sentence $\mathcal{S}_6$:

"If you have stomach upset while taking this medication, take it with food, milk, or an antacid".

Predicates: The predicates extracted by both human and machine match.

A = you have stomach upset
B = taking this medication
C = take it with food, milk, or an antacid

Formal Expressions:

Format	Human output	Machine output
LingSem	*If* (*while* $(B, A), C$)	*If* (*while* $(B, A), C$)
LTL	$\mathcal{G}((A \wedge B) \rightarrow \mathcal{F}(C))$	$\mathcal{G}((FB \leftrightarrow FA) \rightarrow \mathcal{F}\ C)$

Both human and machine LTL outputs are wrong.

Discussion: We notice the *while* relation nested within *if*. The participant treated *while* as the logical *and*, which suits the intended meaning. However, we do not expect to see the *eventually* ($\mathcal{F}$) operator because it allows traces that contradict the sentence, e.g., $t_1^6 = \{A, B, \neg C\} \rightarrowtail \{A, B, \neg C\} \rightarrowtail \{\neg A, \neg B, C\}$. We would not like to see the patient not taking the antacid (not satisfying C) during A and B but doing it later after A and B are over. Instead, we would like the same state where A and B are true to have C true. Thus, we expect the expression $\mathcal{G}((A \wedge B) \rightarrow C)$.

Regarding the machine's LTL output, it stems from a direct translation of the *while* and *if* LingSem relations as dictated by the LingSem-to-LTL mapping (Module 5). The resulting output from the

machine is wrong. The mapping dictates a translation of *while* that follows the semantics of the *Co-existence*(A, B) process template (which translates to $FA \leftrightarrow FB$). However, the use of *while* in this particular sentence does not imply that semantics, but rather implies, as mentioned above, $A \wedge B$.

A.5 Sentence $\mathcal{S}_7$:

"The dosage is based on your medical condition and response to treatment".

Predicates by Machine: None. The machine found no dependency patterns in this sentence.

Predicates by Human:

A = The dosage
B = your medical condition
C = response to treatment

Formal Expressions:

Format	Human output	Machine output
LingSem	*Based-on* (*and* $(B, C), A$)	—
LTL	$\mathcal{G}((B \wedge C) \rightarrow \mathcal{F}(A))$	—

Discussion: This sentence has no syntactic pattern which the machine can detect. The semantics of this sentence relies on the semantics of its main verbal phrase *is based on*. The participant captured this in her LingSem expression by crafting a LingSem relation called *Based-on*. However, this relation is not correct since it does not relate two predicates. In fact, by definition (Section 2.1.1), the entire sentence is a single predicate and contains no sub-clauses. Based on that, we argue that the LingSem expression is incorrect and, by extension, so is the LTL expression.

Since the machine has not adopted verb semantics, it can infer no meaning from this sentence.

A.6 Sentence $\mathcal{S}_8$:

"To reduce your risk of stomach bleeding and other side effects, take this medication at the lowest effective dose for the shortest possible time".

Predicates by Machine: None. The machine found no dependency patterns in this sentence.

A = reduce your risk of stomach bleeding and other side effects
B = take this medication at the lowest effective dose for the shortest possible time

Formal Expressions:

Format	Human output	Machine output
LingSem	*To* (A, B)	—
LTL	$\mathcal{G}(B \rightarrow \mathcal{F}(A))$	—

Discussion: Here, as in Sentences 4 and 5, we notice a temporal constraint that should ideally be captured by the formal output. While the participant found a LingSem relation *To*, the machine did not detect the linguistic pattern because it is not defined in its list of dependency patterns. We detail this below. Regarding the LTL expression, it is arguable whether the meaning behind the sentence is reflected in the effect of B on A. If the effect is immediate, then the expressions should be $\mathcal{G}(B \rightarrow A)$. If the effect is in the next time step, then the expression should be $\mathcal{G}(B \rightarrow \mathcal{X}(A))$ as the participant chose for Sentence $\mathcal{S}_5$. If the effect is in some time in the future, then the chosen expression $\mathcal{G}(B \rightarrow \mathcal{F}(A))$ is the correct one.

The auxiliary *to*. In Sentences $\mathcal{S}_8$ and $\mathcal{S}_{12}$, we do not detect the *to* as a marker to an adverbial clause. That is because the dependency trees of both these sentences show *to* as an auxiliary `aux` to the verb. We cannot rely on that to make a rule to detect such a pattern `VERB` $\xrightarrow{advcl}$ `VERB` $\xrightarrow{aux}$ *to*. This is unreliable because an auxiliary carries no semantics by itself, unlike marker words such as *if* and *while*. In the case of an auxiliary, the meaning of the adverbial clause completely relies on the verb of the clause. Once verb semantics are adopted, then such a case can be covered. When modifying the sentence as follows: "Take this medication at the lowest effective dose for the shortest possible time *so that you* reduce your risk of stomach bleeding and other side effects", the machine detected a generic adverbial clause pattern and produced the correct LingSem tree as seen in the output of Sentence $\mathcal{S}_{8_2}$ in Appendix C.

A.7 Sentence $\mathcal{S}_9$:

"Do not increase your dose or take this drug more often than directed by your doctor or the package label".

First Solution by Human:

A = Do not increase your dose
B = take this drug more often than directed by your doctor
C = the package label

Formal Expressions:

Format	Human output	Machine output
LingSem	*or* $(A, or(B, C))$	—
LTL	$A \vert B \vert C$	

Second Solution:

Predicates: The predicates extracted by both human and machine match.

A = Do not increase your dose
B = take this drug more often than directed by your doctor or the package label

In addition, the machine defined the following predicate:

$\bar{A}$ = Do increase your dose

Formal Expressions:

Format	Human output	Machine output
LingSem	$or\ (A, B)$	$or\ (not(\bar{A}), B)$
LTL	$A \vert B$	$\neg\bar{A} \vert B$

Discussion: The outputs here are considered a match between human and machine modulo the negation detected by the machine. Both outputs are correct.

A.8 Sentence $\mathcal{S}_{10}$:

"For ongoing conditions such as arthritis, continue taking this medication as directed by your doctor".

Predicates by Human:

A = ongoing conditions such as arthritis
B = continue taking this medication as directed by your doctor

Formal Expressions:

Format	Human output	Machine output
LingSem	$for\ (A, B)$	—
LTL	$\mathcal{G}(A \rightarrow \mathcal{F}(B))$	—

Deep detected, but disconnected patterns. In this sentence, the machine states that it detects at least one deep dependency pattern, but cannot recognize the relation(s) between the root and the deep nodes. Therefore, it cannot proceed. The deep pattern detected is the `adv-clause generic` pattern which matches the adverbial clause "as directed by your doctor". The reason it cannot proceed with any further processing of the deep pattern is because of the unrecognized dependency relations running from the root of the tree to the deep pattern. These unrecognized relations may carry semantics that supersede, nullify or influence the semantics carried by the deep pattern. We see this phenomenon again in Sentence $\mathcal{S}_{15}$.

A.9 Sentence $\mathcal{S}_{11}$:

"When ibuprofen is used by children, the dose is based on the child's weight".

The predicates by both human and machine match.

Predicates:

A = ibuprofen is used by children
B = the dose is based on the child's weight

Formal Expressions:

Format	Human output	Machine output
LingSem	*When* (A, B)	*When* (A, B)
LTL	$\mathcal{G}(B \rightarrow \mathcal{F}(A))$	$\mathcal{G}(A \rightarrow B)$

Discussion: Here, the participant treated *when* as the *Response* process template. The machine treats *when(A,B)* as "*whenever A is true, B is also true*". Applying this to the sentence, whenever Ibuprofen is used by children, the dose is based on the child's weight. That is a correct interpretation. On the other hand, the participant's LTL is not correct because it allows a trace as $\{B \wedge \neg A\} \rightarrowtail \{\neg A\} \rightarrowtail \{A\}$. In this trace, A is false at the time when B is true and it is also false in the next time unit; A is only true after at least one time unit. This is clearly a wrong interpretation of the sentence. Therefore, in this case, we pick the machine's output to be more accurate.

A.10 Sentence $\mathcal{S}_{12}$:

"Read the package directions to find the proper dose for your child's weight".

The machine found no dependency patterns in this sentence.

Predicates by Human:

A = Read the package directions
B = find the proper dose for your child's weight

Formal Expressions:

Format	Human output	Machine output
LingSem	*to* (B, A)	—
LTL	$\mathcal{G}(A \rightarrow \mathcal{X}(B))$	—

A.11 Sentence $\mathcal{S}_{14}$:

"For certain conditions (such as arthritis), it may take up to two weeks of taking this drug regularly until you get the full benefit".

Predicates by Human:

A = certain conditions (such as arthritis)
B = it may take up to two weeks of taking this drug regularly
C = you get the full benefit

Predicates by Machine:

D = For certain conditions (such as arthritis), it may take up to two weeks of taking this drug regularly
C = you get the full benefit

Formal Expressions:

Format	Human output	Machine output
LingSem	*until* $(C, for\ (A, B))$	*until* (C, D)
LTL	$\mathcal{G}((A \wedge B) \rightarrow\ \mathcal{U}\ (C))$	$\mathcal{G}(D\ \mathcal{U}\ C)$

Discussion: We see a temporal constraint represented in *two weeks*. In this case, the LTL output by the participant is syntactically wrong because she uses the Until ($\mathcal{U}$) operator as a unary operator while it is binary. On the other hand, the machine's output states that: it is always the case that it may take two weeks for the medicine to show the full benefit, but when the time comes when it does show full benefit, then it may not take two weeks any more. So the Until ($\mathcal{U}$) operator here works well to convey the intended meaning accurately.

A.12 Sentence $\mathcal{S}_{15}$:

"If you are taking this drug 'as needed' (not on a regular schedule), remember that pain medications work best if they are used as the first signs of pain occur".

Predicates by Human:

A = you are taking this drug 'as needed' (not on a regular schedule)
B = pain medications work best
C = they are used as the first signs of pain occur

Predicates by Machine:

A = you are taking this drug 'as needed' (not on a regular schedule)
D = remember that pain medications work best if they are used as the first signs of pain occur

Formal Expressions:

Format	Human output	Machine output
LingSem	*if* $(A, if\ (C, B))$	*if* (A, B)
LTL	$\mathcal{G}(A \rightarrow \mathcal{F}(C \rightarrow B))$	$\mathcal{G}((A \rightarrow\ \mathcal{F}\ D))$

Discussion: We notice from the predicates of the participant that none of them include the verb *remember* while the object of the verb (the thing to remember) has been broken down into two

predicates B and C. This is because she embedded the semantics of the verb *remember* in her LTL formula. Note that when considering the semantics of *remember*, she decides that the statement "*Remember* X" simply implies X. This allows her to further process the inner *if* construct that is the object of the verb *remember*. The machine could not do the same because it has no way of judging the semantics of *remember*. Nevertheless, the participant's assumption about the semantics of *remember* is not correct. The formula f_{15h} allows traces such as $t_1^{15} = \{\neg A, \neg B, C\} \rightarrowtail \{A\}$. In the single state of this trace, the statement $C \rightarrow B$ does not hold. However, in the sentence, the object of *remember* is a fact that is temporally independent of A and should certainly hold true before A occurs as well as remain true afterwards. The machine does not make this mistake. Its output only dictates that D, the remembrance, should eventually occur.

Therefore, while the human output tries to capture more elements, it commits a subtle mistake whereas the machine output is correct.

A.13 Sentence $\mathcal{S}_{16}$:

"If you wait until the pain has worsened, the medication may not work as well".

The predicates by both human and machine match.

Predicates:

A = you wait
B = the pain has worsened
C = the medication may not work as well

Formal Expressions:

Format	Human output	Machine output
LingSem	*if* (*until* $(B, A), C)$	*if* (*until* $(B, A), C)$
LTL	$\mathcal{G}((A \mathcal{U} B) \rightarrow (\mathcal{F}\ C))$	$\mathcal{G}((A \mathcal{U} B) \rightarrow (\mathcal{F}\ C))$

A.14 Sentence $\mathcal{S}_{17}$:

"If your condition persists or worsens, or if you think you may have a serious medical problem, get medical help right away".

The predicates by both human and machine match.

Predicates: The predicates extracted by both human and machine match. Similar to $\mathcal{S}_{13}$, there is a small discrepancy for predicate C, which is labeled C_h and C_m for human and machine respectively.

A = your condition persists
B = worsens
C_h = you think you may have a serious medical problem

C_m = if you think you may have a serious medical problem
D = get medical help right away

Formal Expressions:

Format	Human output	Machine output
LingSem	*if* $(or\ (C, or\ (A, B)), D)$	*if* $(or\ (C, or\ (A, B)), D)$
LTL	$\mathcal{G}((A \vee B \vee C) \rightarrow \mathcal{X}(D))$	$\mathcal{G}((A \vee (B \vee C)) \rightarrow \mathcal{F}D)$

Discussion: The difference between the two LTL outputs is that the participant used the *Next* ($\mathcal{X}$) operator while the machine used the *eventually* ($\mathcal{F}$) operator as dictated by the LingSem-to-LTL mapping (Module 5). The reason for the participant's choice is that she interpreted the words "right away" sensing the urgency of predicate D. As a consequence the machine's LTL allows traces such as $\{A \wedge \neg D\} \rightarrowtail \{\neg D\} \rightarrowtail \{D\}$ while the human output does not. For this reason, we conclude that the machine output is less accurate whereas the human output is correct.

A.15 Sentence $\mathcal{S}_{18}$:

"If you are using the nonprescription product to treat yourself or a child for fever or pain, consult the doctor right away if fever worsens or lasts more than 3 days, or if pain worsens or lasts more than 10 days".

Here, the human participant made the same change as with Sentence $\mathcal{S}_{17}$.

The predicates by both human and machine match. Similar to $\mathcal{S}_{13}$, there is a small discrepancy for predicate E, which is labeled E_h and E_m for human and machine respectively.

Predicates:

A = you are using the nonprescription product to treat yourself or a child for fever or pain
B = consult the doctor right away
C = fever worsens
D = lasts more than 3 days
E_h = pain worsens
E_m = if pain worsens
H = lasts more than 10 days

Formal Expressions:

Format	Human output	Machine output
LingSem	*if* $(A, if\ (or\ (C, D, E, H), B))$	*if* $(A, if\ (or\ (C, or\ (D, or\ (E, H))), B))$
LTL	$\mathcal{G}(((C \vee D \vee E \vee H)\mathcal{M}A) \rightarrow \mathcal{X}B)$	$\mathcal{G}(A \rightarrow\ \mathcal{F}\ (\mathcal{G}((C \vee (D \vee (E \vee H))) \rightarrow\ \mathcal{F}\ B)))$

Discussion: We see that the LingSem expressions match but the LTL ones differ. In this case, the machine's output, simply dictated by the LingSem-to-LTL mapping (Module 5), states that if you are using the product, then, if, at a future state, any of these four symptoms occur (C or D or

E or H), then eventually consult your doctor. The only part of this that is not accurate is that the urgency of calling the doctor was not conveyed, similar to the output of Sentence $\mathcal{S}_{17}$. The human output captures that in the use of the *Next* ($\mathcal{X}$) operator. However, while the use of the *Strong release* ($\mathcal{M}$) operator is correct, the construct used by the machine to express the relation between A and the *or*-conjuncts would suffice. We argue that the human's output is more accurate than the machine's output.

Appendix B

Dependency Trees of Ibuprofen Usage Text

This appendix shows the dependency trees of sentences of the Ibuprofen text as generated by spaCy version 3.0.7 using the spaCy pipeline package `en-core-web-trf` version 3.1.0.

Figure B.1: Dependency tree of $\mathcal{S}_1$.

Figure B.2: Dependency tree of $\mathcal{S}_2$.

Figure B.3: Dependency tree of $\mathcal{S}_3$.

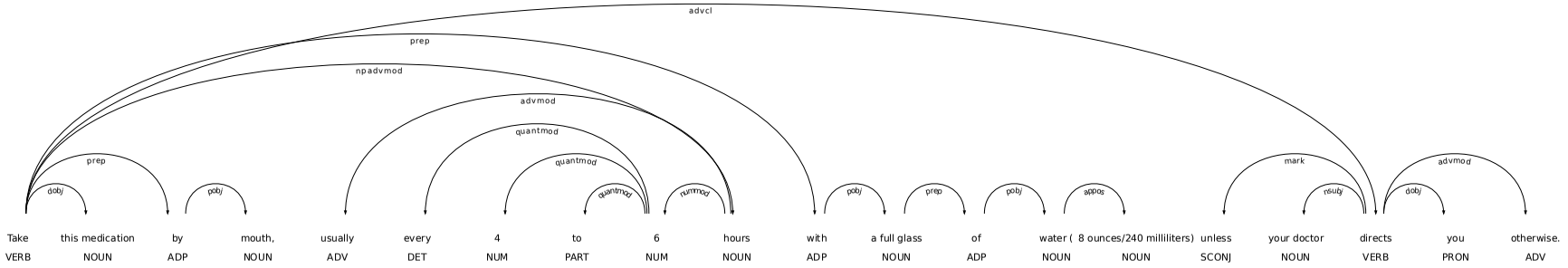

Figure B.4: Dependency tree of $\mathcal{S}_4$.

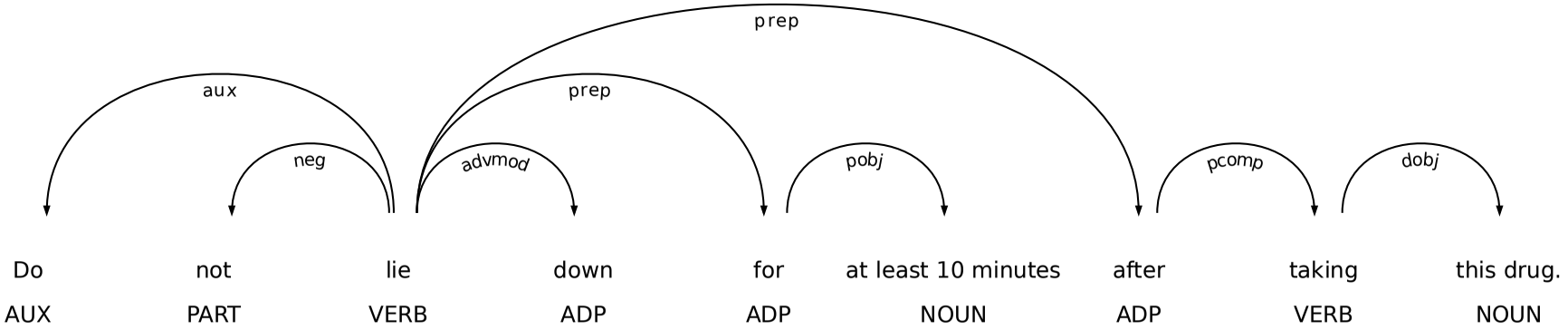

Figure B.5: Dependency tree of $\mathcal{S}_5$.

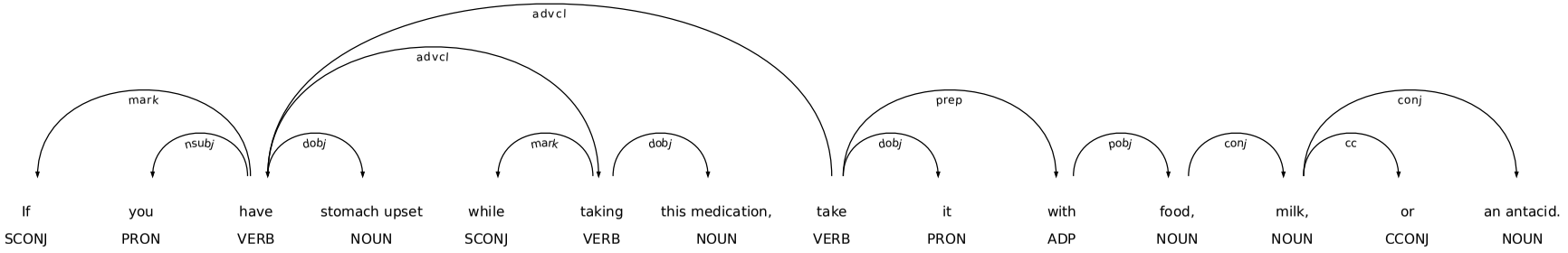

Figure B.6: Dependency tree of $\mathcal{S}_6$.

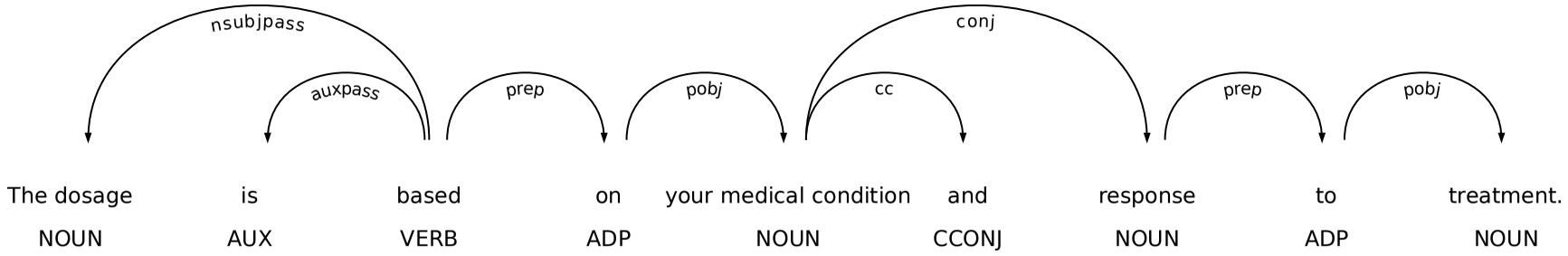

Figure B.7: Dependency tree of $\mathcal{S}_7$.

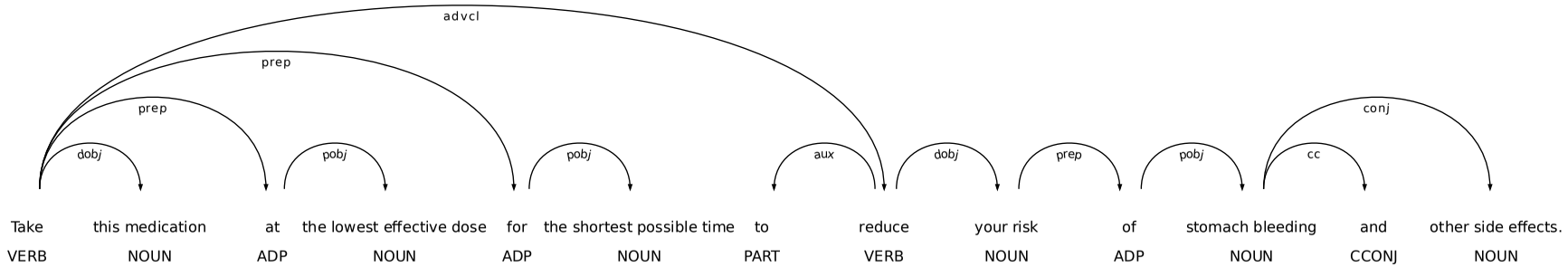

Figure B.8: Dependency tree of $\mathcal{S}_8$.

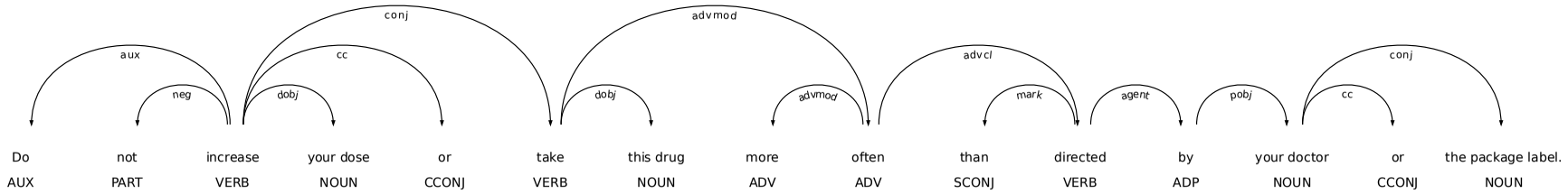

Figure B.9: Dependency tree of $\mathcal{S}_9$.

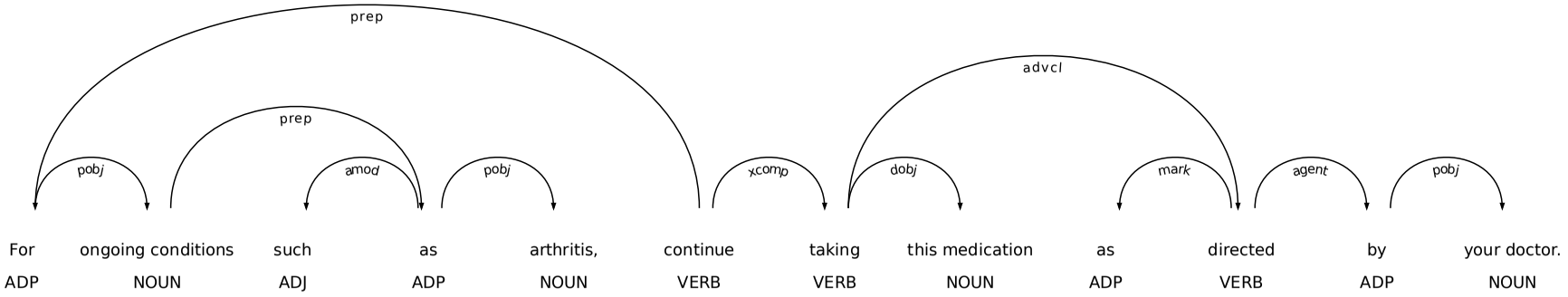

Figure B.10: Dependency tree of $\mathcal{S}_{10}$.

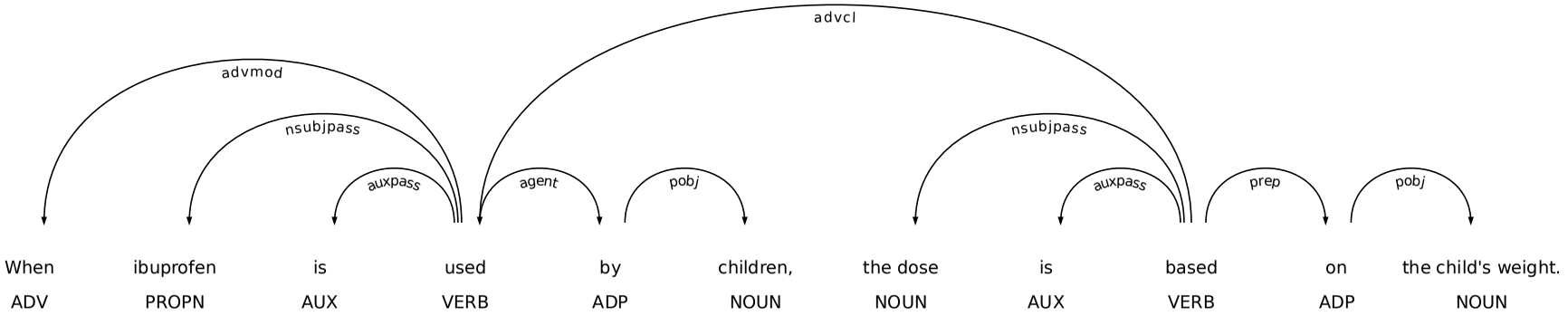

Figure B.11: Dependency tree of $\mathcal{S}_{11}$.

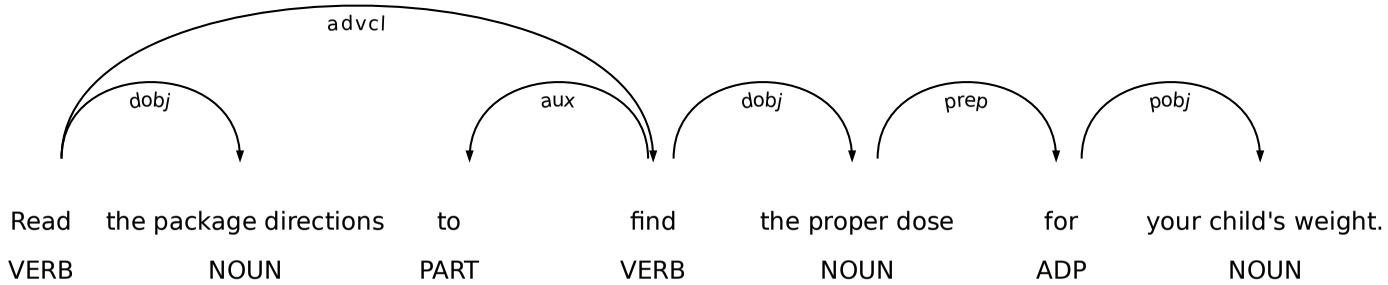

Figure B.12: Dependency tree of $\mathcal{S}_{12}$.

Figure B.13: Dependency tree of $\mathcal{S}_{13}$.

Figure B.14: Dependency tree of $\mathcal{S}_{14}$.

Figure B.15: Dependency tree of $\mathcal{S}_{15}$.

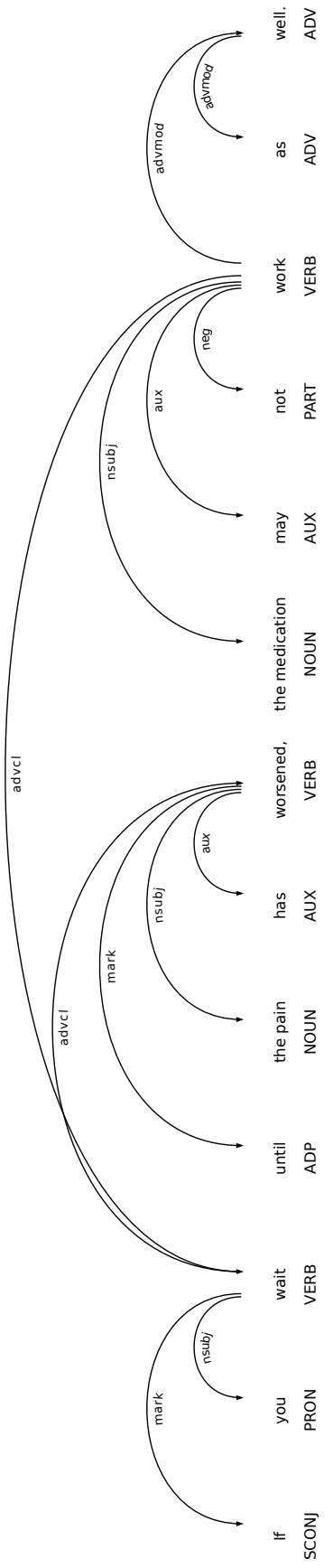

Figure B.16: Dependency tree of $\mathcal{S}_{16}$.

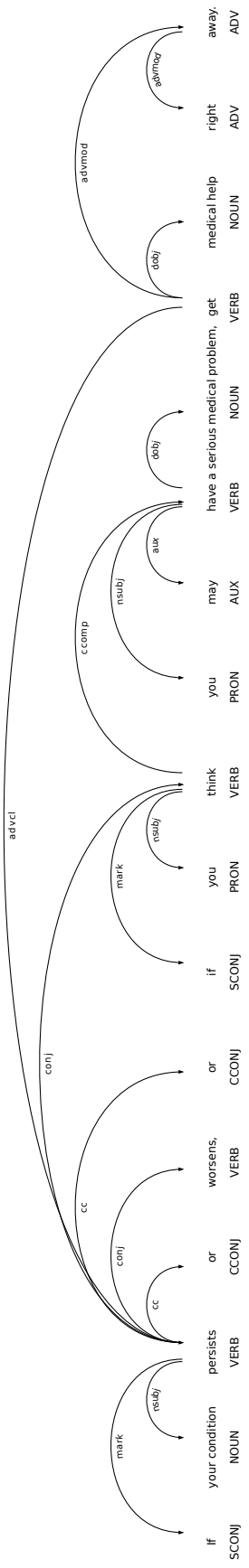

Figure B.17: Dependency tree of $\mathcal{S}_{17}$.

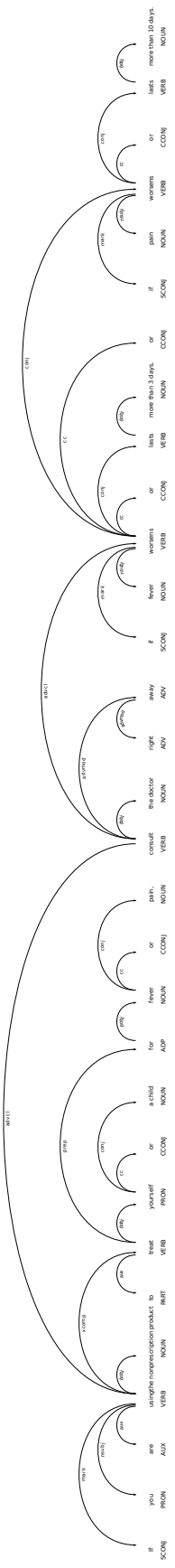

Figure B.18: Dependency tree of $\mathcal{S}_{18}$.

Appendix C

Output of Python Tool

This appendix shows the raw output of the Python tool developed for this thesis. The tool is executed on the text of Ibuprofen usage.

```
Sentence 1: If you are taking the over-the-counter product, read all directions on the
    product package before taking this medication.

Predicates:
A = you are taking the over-the-counter product
B = read all directions on the product package
C = taking this medication

LingSem Tree:
Root: if.
|-- if-clause: you are taking the over-the-counter product,
‘-- then-clause: before.
   |-- main-clause: read all directions on the product package
   ‘-- before-clause: taking this medication.

LingSem String:
if( A , before( C , B ))

LTL Formula:
G ( A -> F ( (~ C) W B ))
```

```
Sentence 2: If your doctor has prescribed this medication, read the Medication Guide provided
    by your pharmacist before you start taking ibuprofen and each time that you get a refill.

Predicates:
A = your doctor has prescribed this medication
B = read the Medication Guide provided by your pharmacist
C = you start taking ibuprofen
```

```
D = each time that you get a refill

LingSem Tree:
Root: if.
|-- if-clause: your doctor has prescribed this medication,
‘-- then-clause: before.
    |-- main-clause: read the Medication Guide provided by your pharmacist
    ‘-- before-clause: and.
        |-- conjunct-1: you start taking ibuprofen.
        ‘-- conjunct-2: each time that you get a refill.

LingSem String:
if(A, before(and(C, D), B))

LTL Formula:
G ( A -> F ( ~ (C & D ) W B ))
```

```
Sentence 3: If you have any questions, ask your doctor or pharmacist.

Predicates:
A = you have any questions
B = ask your doctor or pharmacist

LingSem Tree:
Root: if.
|-- if-clause: you have any questions.
‘-- then-clause: ask your doctor or pharmacist.

LingSem String:
if(A, B)

LTL Formula:
G (A -> F B )
```

```
Sentence 4: Take this medication by mouth, usually every 4 to 6 hours with a full glass of
     water (8 ounces/240 milliliters) unless your doctor directs you otherwise.

Predicates:
A = Take this medication by mouth, usually every 4 to 6 hours with a full glass of water (8
     ounces/240 milliliters)
B = your doctor directs you otherwise

LingSem Tree:
Root: unless.
|-- unless-clause: your doctor directs you otherwise.
```

```
‘-- main-clause: Take this medication by mouth, usually every 4 to 6 hours with a full glass
    of water (8 ounces/240 milliliters)

LingSem String:
unless(B, A)

LTL Formula:
A & G ( (B -> F ( ~ A ) ) )
_____________________________________________________________________________________

Sentence 5: Do not lie down for at least 10 minutes after taking this drug.

Predicates:
A = Do lie down for at least 10 minutes
B = taking this drug

LingSem Tree:
Root: after.
|-- after-clause: taking this drug
‘-- main-clause: not.
   ‘-- operand: Do lie down for at least 10 minutes

LingSem String:
after(B, not(A))

LTL Formula:
A W B
_____________________________________________________________________________________

Sentence 6: If you have stomach upset while taking this medication, take it with food, milk,
    or an antacid.

Predicates:
A = you have stomach upset
B = taking this medication
C = take it with food, milk, or an antacid

LingSem Tree:
Root: if.
|-- then-clause: take it with food, milk, or an antacid.
‘-- if-clause: while.
    |-- occurrent-1: taking this medication.
    ‘-- occurrent-2: you have stomach upset.

LingSem String:
if(while(B, A), C)
```

```
LTL Formula:
G( ( F B <-> F A) ) -> F C )
```

```
Sentence 7: The dosage is based on your medical condition and response to treatment.

No dependency patterns matched.
```

```
Sentence 8: To reduce your risk of stomach bleeding and other side effects, take this
     medication at the lowest effective dose for the shortest possible time.

No dependency patterns matched.
```

```
Sentence 8_2: Take this medication at the lowest effective dose for the shortest possible
     time so that you reduce your risk of stomach bleeding and other side effects.

Predicates:
A = Take this medication at the lowest effective dose for the shortest possible time
B = you reduce your risk of stomach bleeding and other side effects

LingSem Tree:
Root: so that.
|-- adv-clause: you reduce your risk of stomach bleeding and other side effects.
‘-- main-clause: take this medication at the lowest effective dose for the shortest possible
     time.

LingSem String:
so_that(B,A)

LTL Formula:
No mapping found for LingSem relation ’so_that’.
```

```
Sentence 9: Do not increase your dose or take this drug more often than directed by your
     doctor or the package label.

Predicates:
A = Do increase your dose
B = take this drug more often than directed by your doctor or the package label

LingSem Tree:
Root: or.
|-- conjunct-2: take this drug more often than directed by your doctor or the package label.
‘-- conjunct-1: not.
   ‘-- operand: Do increase your dose.
```

```
LingSem String:
or(not(A),B)

LTL Formula:
((~ A) | B))
```

```
Sentence 10: For ongoing conditions such as arthritis, continue taking this medication as
     directed by your doctor.

Warning: At least one dependency pattern was matched that has at least one unrecognized
     ancestor dependency relation.
```

```
Sentence 11: When ibuprofen is used by children, the dose is based on the child's weight.

Predicates:
A = ibuprofen is used by children
B = the dose is based on the child's weight

LingSem Tree:
Root: when.
|-- when-clase: ibuprofen is used by children.
'-- main-clause: the dose is based on the child's weight.

LingSem String:
when(A,B)

LTL Formula:
G( A -> B )
```

```
Sentence 12: Read the package directions to find the proper dose for your child's weight.

No dependency patterns matched.
```

```
Sentence 13: Consult the pharmacist or doctor if you have questions or if you need help
     choosing a nonprescription product.

Predicates:
A = Consult the pharmacist or doctor
B = you have questions
```

```
C = if you need help choosing a nonprescription product

LingSem Tree:
Root: if.
|-- then-clause: Consult the pharmacist or doctor.
‘-- if-clause: or.
    |-- conjunct-1: you have questions.
    ‘-- conjunct-2: if you need help choosing a nonprescription product.

LingSem String:
if(or(B, C), A)

LTL Formula:
G ( ( B | C ) -> ( F A ) )
```

```
Sentence 14: For certain conditions (such as arthritis), it may take up to two weeks of
     taking this drug regularly until you get the full benefit.

Predicates:
A = For certain conditions (such as arthritis), it may take up to two weeks of taking this
     drug regularly
B = you get the full benefit

LingSem Tree:
Root: until.
|-- stopper: you get the full benefit.
‘-- continuous-activity: For certain conditions (such as arthritis), it may take up to two
     weeks of taking this drug regularly.

LingSem String:
until(B, A)

LTL Formula:
G ( B U A )
```

```
Sentence 15: If you are taking this drug "as needed" (not on a regular schedule), remember
     that pain medications work best if they are used as the first signs of pain occur.

Warning: At least one dependency pattern was matched that has at least one unrecognized
     ancestor dependency relation.

Predicates:
A = you are taking this drug "as needed" (not on a regular schedule
B = remember that pain medications work best if they are used as the first signs of pain occur
```

```
LingSem Tree:
Root: if.
|-- if-clause: you are taking this drug "as needed" (not on a regular schedule.
‘-- then-clause: remember that pain medications work best if they are used as the first signs
    of pain occur.

LingSem String:
if(A,B)

LTL Formula:
G ( (A -> F B ) )
```

```
Sentence 16: If you wait until the pain has worsened, the medication may not work as well.

Predicates:
A = you wait
B = the pain has worsened
C = the medication may work as well

LingSem Tree:
Root: if.
‘-- if-clause: until.
|   |-- stopper: the pain has worsened.
|   ‘-- continuous-activity: you wait.
‘-- then-clause: not
   ‘-- operand: the medication may work as well.

LingSem String:
if(until(B, A), C)

LTL Formula:
G ( ( A U B ) -> ( F C ) )
```

```
Sentence 17: If your condition persists or worsens, or if you think you may have a serious
    medical problem, get medical help right away.

Predicates:
A = your condition presists
B = worsens
C = if you think you may have a serious medical problem
D = get medical help right away

LingSem Tree:
Root: if.
|-- then-clause: get medical help right away.
```

```
‘-- if-clause: or.
    |-- conjunct-1: your condition persists.
    ‘-- conjunct-2: or.
      |-- conjunct-1: worsens.
      ‘-- conjunct-2: if you think you may have a serious medical problem.

LingSem String:
if(or(A, or(B,C)), D)

LTL Formula:
G ( (A | (B | C)) -> F D )
```

```
Sentence 18: If you are using the nonprescription product to treat yourself or a child for
     fever or pain, consult the doctor right away if fever worsens or lasts more than 3 days,
     or if pain worsens or lasts more than 10 days.

Predicates:
A = you are using the nonprescription product to treat yourself or a child for fever or pain
B = consult your doctor right away
C = fever worsens
D = lasts more than 3 days
E = if pain worsens
H = lasts more than 10 days

LingSem Tree:
Root: if.
|-- if-clause: you are using the nonprescription product to treat yourself or a child for
     fever or pain.
‘-- then-clause: if.
   |-- then-clause: consult your doctor right away
   ‘-- if-clause: or.
      |-- conjunct-1: fever worsens
      ‘-- conjunct-2: or.
         |-- conjunct-1: lasts more than 3 days.
         ‘-- conjunct-2: or.
            |-- conjunct-1: if pain worsens.
            ‘-- conjunct-2: lasts more than 10 days.

LingSem String:
if(A, if(or(C, or(D, or(E, H))), B))

LTL Formula:
G (A -> F ( G ( ( C | ( D | ( E | H ) ) ) -> F B ) ))
```
