Formalizing Contract Refinements Using a Controlled Natural Language

Regan Meloche

Thesis submitted to the University of Ottawa
in partial Fulfillment of the requirements for the
Master of Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

© Regan Meloche, Ottawa, Canada, 2023

Abstract

The formalization of natural language contracts can make the prescriptions found in these contracts more precise, promoting the development of smart contracts, which are digitized forms of the documents where the monitoring and execution can be partially automated. Full formalization remains a difficult problem, and this thesis makes steps towards solving this challenge by focusing on a narrow sub-problem of formalizing contract *refinements*. We want to allow a contract author to customize a contract template, and automatically convert the resulting contract to a formal specification language called SYMBOLEO, created specifically for the legal contract domain. The hope is that research towards *partial* formalization can be useful on its own, as well as useful towards the *full* formalization of contracts.

The main questions addressed by this thesis involve asking what linguistic forms these refinements will take. Answering these questions involves both linguistic analysis and empirical analysis on a set of real contracts to construct a controlled natural language (CNL). This language is expressive and natural enough to be adopted by contract authors, and it is precise enough that it can reliably be converted into the proper formal specification. We also design a tool, SYMBOLEONLP, that demonstrates this functionality on realistic contracts. This involves ensuring that the contract author can input contract refinements that adhere to our CNL, and that the refinements are properly formalized with SYMBOLEO.

In addition to contributing an evidence-based CNL for contract refinements, this thesis also outlines a very clear methodology for constructing this CNL, which may need to go through iterations as requirements change and as the SYMBOLEO language evolves. The SYMBOLEONLP tool is another contribution, and is designed for iterative improvement. We explore a number of potential areas where further NLP techniques may be integrated to improve performance, and the tool is designed for easy integration of these modules to adapt to emerging technologies and changing requirements.

Acknowledgements

I am grateful to my co-supervisors, Prof. Daniel Amyot and Prof. John Mylopoulos, for their academic guidance throughout my studies. Through their wisdom, I learned how to navigate the world of graduate studies, approach a thesis by asking the right questions, and balance my personal interests with relevant research topics.

Other collaborators in our research group include Prof. Luigi Logrippo, Prof. Marco Roveri, Amal Anda, Sofana Alfuhaid, and Daniel Sousa-Dias, and I would like to thank them for providing weekly feedback and insights from their related areas of expertise. I would also like to thank Prof. Mehrdad Sabetzadeh and Prof. Diana Inkpen for agreeing to be examiners for this thesis, for their feedback and insight in the early stages of writing the thesis, and also for the engaging courses that I took throughout my studies. Finally, this work has been impacted by, and builds on, the work of Aidin Rasti, Alireza Parvizimosaed, and Sepehr Sharifi.

This work was partially funded by an NSERC Discovery Grant #610877 titled *Engineering Requirements for Cyber-Physical Systems* and by the ORF-RE project *CyPreSS: Software Techniques for the Engineering of Cyber-Physical Systems*.

Dedication

I dedicate this thesis to Atidthan, who has been a constant source of support and encouragement as I follow this dream. This is also dedicated to my parents, Marty and Laurie, for always pushing me to realize my full potential.

Table of Contents

List of Tables x

List of Figures xi

List of Listings xiii

Glossary xv

1 Introduction 1

1.1 Motivation . . . 1
1.2 Problem Statement . . . 3
1.3 Research Questions (RQs) . . . 6
1.4 Contributions . . . 7
1.5 Structure of Thesis . . . 8

2 Preliminaries 9

2.1 Symboleo . . . 9
2.1.1 Legal Contract Ontology . . . 9
2.1.2 Sample Contract . . . 12
2.1.3 Main Components . . . 15
2.1.4 Syntax of SYMBOLEO . . . 17
2.1.5 Semantics of SYMBOLEO . . . 21
2.2 Natural Language Processing . . . 23
2.2.1 Formalization . . . 24
2.2.2 Machine Learning (ML) Approaches . . . 24
2.2.3 Linguistic Approaches . . . 26
2.2.4 Controlled Natural Language . . . 28

2.3 Linguistics . . . 29
2.3.1 Sentences, Clauses, Phrases . . . 29
2.3.2 Prepositions . . . 30
2.3.3 Adjuncts . . . 31
2.4 Summary . . . 32

3 Related Work **34**
3.1 Formal Representations . . . 34
3.1.1 Legal Ontologies . . . 34
3.1.2 Smart Contract Specification Languages . . . 37
3.1.3 Relationship with SYMBOLEO . . . 38
3.2 Controlled Natural Languages . . . 38
3.2.1 CNL Design Considerations . . . 38
3.2.2 Attempto Controlled English . . . 40
3.2.3 CNLs for the Legal Domain . . . 40
3.2.4 CNLs for Requirements Engineering . . . 41
3.2.5 Formal Languages . . . 42
3.3 Event Specification . . . 42
3.3.1 FrameNet . . . 42
3.3.2 ACE . . . 43
3.3.3 Frame-based Event Specification . . . 44
3.4 Formalization . . . 45
3.4.1 Manual Efforts . . . 46
3.4.2 Linguistic-based Approaches . . . 47
3.4.3 Other Linguistics-based Approaches . . . 49
3.4.4 ML-based Approaches . . . 50
3.4.5 Bridging Linguistics and ML Approaches . . . 50
3.4.6 Application to SYMBOLEO . . . 51
3.5 Contract Customization . . . 51
3.5.1 GaiusT and ContracT . . . 52
3.5.2 Contract CNL to Deontic Logic . . . 53
3.5.3 Custom Contract Templates with a CNL . . . 54
3.5.4 Relevant Work from Requirements Engineering . . . 55
3.6 Summary . . . 56

4 Constructing a CNL **58**
4.1 Templating Approach . . . 58
4.1.1 Comparison with FITB . . . 58
4.1.2 Framing the RQs . . . 59
4.2 RQ1: Choosing Symboleo Operations . . . 60
4.2.1 Listing Potential Operations . . . 61
4.2.2 Principles of Exclusion . . . 63
4.2.3 Refining and Broadening Predicates . . . 65
4.3 RQ2: Finding NL Correspondence . . . 67
4.3.1 CNL Design . . . 68
4.3.2 Principle of Integrity and the Adjunct . . . 69
4.3.3 Specifying Building Blocks . . . 70
4.3.4 Predicate Refinements . . . 75
4.3.5 Adding Predicates . . . 79
4.3.6 Domain Model Updates . . . 81
4.3.7 Summary of Operations . . . 83
4.4 RQ3: An Evidence-based CNL . . . 85
4.4.1 Outline of RQ3 . . . 85
4.4.2 Building the Dataset . . . 86
4.4.3 CNL Construction Example . . . 89
4.4.4 Event Complexity . . . 91
4.4.5 Predicate Refinements . . . 93
4.4.6 Adding Predicates . . . 97
4.4.7 New Patterns and Operations . . . 99
4.4.8 Final Results . . . 101
4.4.9 Precise Formulation of Problem . . . 103
4.5 Evaluation . . . 104
4.5.1 Evaluation Examples . . . 105
4.5.2 Analysis and Threats to Validity . . . 107
4.6 Summary . . . 109

5 Application **110**
5.1 Problem Statement . . . 110
5.1.1 Manual Preparation of the Template . . . 111
5.1.2 Application Requirements . . . 115
5.1.3 Tool Design Decisions . . . 116
5.2 Generating CNL Patterns . . . 118
5.2.1 A Simpler Example . . . 118
5.2.2 Range of Input . . . 120
5.2.3 Mitigating User Errors . . . 122
5.2.4 Assumption of Linguistic Knowledge . . . 122
5.2.5 Sample Refinement . . . 124
5.2.6 Evaluation . . . 128
5.3 Mapping CNL to Symboleo . . . 130
5.3.1 Overview and Algorithm . . . 130
5.3.2 Pattern and Event Extraction . . . 131
5.3.3 Norm Update Extraction . . . 132
5.3.4 Domain Update Extraction . . . 133
5.3.5 Identifying and Filling Event slots . . . 134
5.3.6 Completing the Mapping . . . 136
5.4 Walkthrough . . . 139
5.4.1 P2: Unless EVENT . . . 140
5.4.2 P3: Added Antecedent . . . 142

6 Discussion **144**
6.1 CNL for Contract Refinements . . . 144
6.1.1 RQ1: Identifying Operations and Symboleo . . . 144
6.1.2 RQ2: Semantic Analysis . . . 146
6.1.3 RQ3: Evidence-based Construction of the CNL . . . 147
6.1.4 Classifying our CNL . . . 148
6.1.5 Contributions . . . 149
6.2 Proof-of-Concept Tool using the CNL . . . 151
6.2.1 RQ4: Generating Valid CNL . . . 151
6.2.2 RQ5: Mapping CNL to Operations . . . 152

6.2.3 SymboleoNLP Road Map 153
6.3 Longer-term Road Map 154
6.3.1 Towards Full Formalization 155
6.3.2 Incorporating LLMs 156

7 Conclusion **160**
7.1 Problem Statement 160
7.2 CNL Construction (RQ1–RQ3) 160
7.3 SymboleoNLP (RQ4, RQ5) 162

A Non-Temporal Dynamic Properties **165**

B CNL Generation and Artifacts **167**
B.1 CUAD Filtering 167
B.2 Norm Identification 168
B.3 Pattern Aggregation 170
B.4 Pattern Class Aggregation 171

C CNL Implementation **173**
C.1 Grammar 173
C.2 Pattern Classes 176
C.3 Tree Construction 177

D Pattern Extraction **180**

E Event Extraction Details **182**

F Contract Specification Parameter Processing **185**

List of Tables

2.1 Text from a sample Meat Sale contract [80] 13
2.2 Predicates of Symboleo 19

4.1 Initial CNL mappings 84
4.2 Final CNL mappings 102

5.1 Simplified Rental Agreement Sample (T0) 111
5.2 Broadened Rental Agreement Sample (T) with parameters 113
5.3 Input units for P1 refinement 129
5.4 Input units for P2 refinement 140
5.5 Input units for P3 refinement 142

B.1 Sample of operation/pattern pairings 171

E.1 Input units and values for the sample event 182

List of Figures

1.1 Diagram of our problem structure 5

2.1 Symboleo ontology (from Parvizimosaed et al. [70]) 10
2.2 Diagram showing relationships between predicates and their arguments . 20
2.3 Simple state diagram based on a small subset of the axioms 22
2.4 Full state charts for contract, obligation, and power 23
2.5 Linguistic information extracted from the sentence 'the buyer must pay $100 to the seller' 26
2.6 Dependencies extracted from the sentence 'the buyer must pay $100 to the seller' 27
2.7 Three interpretations of 'before' 31

4.1 Spectrum showing the relationship between complexity and flexibility, as well as where the present work is situated. 59

5.1 Diagram showing inputs and flow of SymboleoNLP 114
5.2 Snippet of Symboleo's XText grammar showing the PointFunction . . . 116
5.3 Simple CNL selection tree 119
5.4 Subset of the input unit selection tree 121
5.5 User selects a parameter to refine 125
5.6 User is presented with all children of the root node 125
5.7 User selects the 'weeks' option for the time unit of the timespan component 126
5.8 User is presented with 3 types of verbs 127
5.9 User enters a dynamic value for the transitive verb 127
5.10 Completed user entry 128
5.11 Screenshot showing 100% code coverage on SymboleoNLP 129
5.12 Updated NL template 137
5.13 Updated Symboleo specification 138

6.1 Example of ChatGPT failing a simple verb-type question 157
6.2 Providing ChatGPT with context of a simple EBNF grammar 158
6.3 ChatGPT correctly identifying a pattern class 159

E.1 SpaCy's parse of the direct object 184

List of Listings

2.1 SYMBOLEO specification of the meat sale contract 13
2.2 Sample of the SYMBOLEO syntax 18
2.3 Grammar for specifying an obligation 27
3.1 Grammar used in ContracT 52
3.2 Grammar used in a template-based approach to contract customization (Tateishi et al. [88]) 54
4.1 Grammar for specifying a standard event 71
4.2 Grammar for specifying a timepoint 73
4.3 Grammar for specifying an interval 74
4.4 Initial working grammar 84
4.5 Final grammar 101
5.1 SYMBOLEO specification of the rental agreement S(T0) 111
5.2 SYMBOLEO specification of the broadened rental agreement S(T) 112
5.3 SYMBOLEO obligation expressed in Python 117
5.4 Simple CNL grammar 118
5.5 CNL input generation algorithm 124
5.6 Sample pattern class test 130
5.7 CustomEvent sample specification 131
5.8 Norm update extractor for the sample pattern class 133
5.9 Operation mapping algorithm 136
5.10 Pattern class handler test sample 138
5.11 Full contract refinement algorithm 139
B.1 First filtering step on the dataset 168
B.2 Identifying potential norms 168
B.3 Identifying refinement heuristics 169
B.4 Aggregating patterns 171
C.1 Representation of the P_BEFORE_S pattern variable 173
C.2 Sample of static variables 174
C.3 Sample of Input Units 174
C.4 Timespan specification 175
C.5 Full SYMBOLEONLP Grammar 175
C.6 Specification of the WITHIN TIMESPAN P_AFTER_W EVENT pattern class . 177
C.7 Code to build a tree structure from a pattern class 178
C.8 Code to merge two trees 178
C.9 Code for building the full grammar tree 179
D.1 Pattern class verification 181

D.2 Full pattern class extraction 181
E.1 Subject specification 182
E.2 Verb specification 183
E.3 Direct object specification 184

Glossary

ACE Automatic Content Extraction

ACE Attempto Controlled English

BERT Bidirectional Encoder Representations from Transformers

CLG Controlled Legal German

CLO Core Legal Ontology

CNL Controlled Natural Language

CSL Contract Specification Language

CSP Contract Specification Parameter

CUAD Contract Understanding Atticus Dataset

DOLCE Descriptive Ontology for Legal and Cognitive Engineering

DSL4SC Domain-Specific Language for Smart Contracts

EBNF Extended Backus–Naur Form

EDGAR Electronic Data Gathering, Analysis, and Retrieval

FITB Fill-in-the-Blank

GDPR General Data Protection Regulation

GPT Generative Pre-Training Transformer

LKIF Legal Knowledge Interchange Format

LLM Large Language Model

LSTM Long-Short-Term Memory

ML Machine Learning

NIR Norm-in-Rete

NL Natural Language

NLP Natural Language Processing

NLTK Natural Language Toolkit

NP Noun Phrase

OWL Web Ontology Language

PENS Precision, Expressiveness, Naturalness, and Simplicity

POS Parts of Speech

PP Prepositional Phrase

RNN Recurrent Neural Network

RQ Research Question

SEC Securities and Exchange Commission

UFO Unified Foundational Ontology

UFO-L UFO-Legal

VP Verb Phrase

Chapter 1

Introduction

This thesis explores the construction of a controlled natural language designed for the formalization of contract customizations, and describes a tool, SymboleoNLP, that applies this formalization approach to a set of realistic contract templates.

1.1 Motivation

Many of our daily affairs, both business and personal, are mediated by normative documents. For example, a government regulation may set limits on how many hours an employee in a certain industry can work per week. A sales contract may specify that a certain quantity of a product must be delivered to a specific address before a due date. A lease agreement may prohibit a tenant from making excessive noise after a certain time of day. All of these documents prescribe behaviour of how one party (employer, supplier, tenant) must act towards another party (employee, buyer, property owner). These documents are often legally binding, so they must be unambiguous, understandable, and sufficiently precise to the parties involved. A lack of these qualities can lead to different interpretations of the prescriptions that may result in costly and time-consuming legal disputes.

An emerging format of normative document is the smart contract [87]. A smart contract is a digital artifact where certain prescriptions and conditions found in traditional contracts are executed and monitored automatically by computer systems. Their growth is largely due to technologies such as distributed ledgers and the Internet-of-Things. Some example use cases of smart contracts include:

- **Food Supply Chain [5, 44, 82]:** Suppose a meat sale contract specifies that the meat being delivered must always be maintained below a certain temperature. If the temperature rises above that threshold, then the quality is compromised, and the buyer may be entitled to a refund. A temperature sensor inside a delivery vehicle connected to the internet can detect the change in temperature and report this to a digital contract, which can then automatically trigger (or cancel) a payment transaction.

- **Service-Level Agreements [1, 77]:** Suppose a software-as-a-service company wants to guarantee its clients that their software remains up and running throughout the entire year with minimal downtime. The contract may specify that they are allowed to have 10 hours of downtime per year before the client is entitled to a discount. Once again, the downtime can be monitored automatically. If it is detected that the total downtime lasts longer than the prescribed amount, the smart contract can trigger an automatic payment or discount.

- **Privacy Policies [42, 59]:** There is a public interest for governments to impose requirements on how tech companies are allowed to handle data from their users. Companies may be restricted from asking for certain types of personal data or from selling user data. Data flows could be monitored to ensure that these companies are complying with data privacy prescriptions. If a breach of contract is detected, then appropriate penalties may be imposed.

One challenge involved in the development of a smart contract is the conversion of a normative document written in a Natural Language (NL), such as English, into code that can be processed by a computer. To illustrate, consider a contrived norm that may be found in a supply agreement: 'The supplier must make reasonable efforts to deliver the goods within 30 days of order placement to the buyer, otherwise they are entitled to a 10% discount.' This sentence may appear straightforward for a human to understand, but it becomes unclear when we start to question the exact meanings of the terms:

- What is the precise meaning of 'reasonable effort'? Is there a specific industry standard that could be referenced instead?

- Does '30 days' include weekends and holidays, or just business days?

- Where exactly should the delivery be made? For example, does the buyer have a warehouse to which the goods should be delivered?

- The 'they' in 'otherwise they are entitled' is somewhat ambiguous. In this context, it is reasonable to think it refers to the buyer, but other cases of pronouns may not be so obvious.

The humans on each side of this contract may have an agreed-upon meaning of each of these terms, or the specific details may be laid out earlier in the contract. An employee at the buyer's company can keep track of these provisions and request appropriate discounts from the seller. However, it may be the case that the humans *do not* have agreed-upon meanings, in which case a legal dispute may arise when the contract terms are tested (e.g., the seller is a day late on delivery, but there was a severe thunderstorm).

The key point is that any specific interpretation of these terms is implicit, relying on industry conventions or common sense. In almost any NL communication, a certain degree of common sense is assumed, and this assumption is a fundamental problem if we want computers to reliably process NL. For a reliable smart contract, we must translate a normative document written in NL into a structured representation that retains the full meaning of the original document. This representation is called a *formal specification*,

and it is achieved by restricting the language that we use to express the information to a language with precisely defined syntax and semantics, leaving no room for ambiguity. This type of representation is called a *specification language*, and the process of converting a NL document to a formal specification is *formalization*. Automating this process is in the domain of Natural Language Processing (NLP), and there are a wide variety of techniques in this field that may help us achieve the goal of partial automation.

In addition to providing the ability to represent normative documents as smart contracts, there are other benefits that come with formalization. If a document is represented in computer code, we can write a series of tests against the code to ensure that it achieves the expected behaviour in a wide variety of scenarios. For example, if a government is creating a benefits program for certain individuals, then simulations could be run to measure the impact that it may have on the entire population as well as different sub-populations[1]. This could be valuable in scenarios where the prescriptions are complex and there are many scenarios to account for [85]. Going a step further than tests and simulations, formalization also allows us to prove certain properties about our document using formal methods. Tests and simulations can be useful, but as Djikstra famously noted, they cannot *prove* the absence of any bugs [23]. Formal methods, however, such as model-checking and theorem-proving, allow us to be mathematically certain about certain behavioural aspects of our code or model, under specific assumptions [70].

Legal contract review also benefits from formalization. Traditionally, this is a process where legal experts carefully review a contract to extract important information for a stakeholder. Since legal documents are often written in complex legalese, this process can be overwhelming for those who are not legal experts. A small business that is signing a contract with a large distributor, or a tenant that is signing a new lease for an apartment, may not have the financial/legal resources to extract information that is important to them. Translating the contract into an unambiguous language can facilitate the process of partially automated contract review, allowing all stakeholders to extract useful information in a convenient representation. It is indeed *possible* to unambiguously specify a contract using a NL such as English, but this possibility is often unrealized due to a reliance on implicit knowledge and common sense. Using a formal specification language forces one to confront the ambiguities inherent in NL and helps ensure that all parties involved have a common interpretation of the contract.

1.2 Problem Statement

Due to the benefits gained from the formalization of normative documents, the automation of this formalization is an active field of study, yet it still remains a hard problem, largely due to the ambiguity inherent in NL as discussed above. A long-term goal in this field would be to take a normative document in NL, and a target specification language, and have the document automatically formalized to that specification language. This is currently over-ambitious, so we focus our efforts on a more manageable component of that goal. We begin by narrowing the domain. There are many types of normative documents (laws,

[1] `https://www.statcan.gc.ca/en/microsimulation/spsdm/spsdm`

government regulations, company policies, contracts, software requirements, etc.), each with unique characteristics. While techniques from one domain may overlap with others, we will focus specifically on the monitoring of legal contracts. The formal specification language SYMBOLEO has been developed specifically for this domain [70, 79], and it will therefore be the focus of this work.

As an example of SYMBOLEO, consider the simple NL obligation: `'The seller shall deliver the goods to the buyer before [DELIVERY_DATE]'`. This would be represented in SYMBOLEO as follows: `O(seller, buyer, true, ShappensBefore(evt_deliver_goods, DELIVERY_DATE))`. The structure and meaning of statements like this will be discussed in detail later on, but for now, we note that SYMBOLEO is a specification language for the legal contract monitoring domain that has a well-defined syntax and semantics.

Rather than focusing our efforts on full formalization of a NL contract, we will focus on formalizing linguistic sub-structures that regularly appear in contracts by using a *template-based* approach. For a given type of contract, an organization may start with a standard template that can be further customized to suit the specific needs of their client. The template only needs to go through a series of minor customizations to be considered complete. Rather than formalizing brand new NL contracts, we can begin with a manually-created formalization, which has semantic correspondence to the NL contract template. The NL template is then customized by a contract author, and the goal is to automatically formalize this newly-customized contract into SYMBOLEO. To do so, we need only to focus on the *changes* that were made to the template, rather than the entire document.

Traditionally, these contract templates may consist of a set of simple blanks to be filled with concepts such as due dates, contracting parties, prices, etc. We will call these simple templates *Fill-in-the-Blank (FITB)* contracts, and we refer back to our example: 'The seller shall deliver the goods to the buyer before [DELIVERY_DATE].' Automated formalization involving simple templates like this is quite trivial, and doesn't make much progress towards full formalization. We are therefore interested in pushing the boundaries of the types of customizations that a contract author might make to a template. Rather than simply filling in the delivery date field on a FITB template with an absolute date value, the author could customize it with a more linguistically complex refinement. For example, suppose the template is as follows: 'The seller shall deliver the goods to the buyer [REFINEMENT]'. The REFINEMENT parameter could be filled with more complex language such as `'before the contract terminates'`, `'within 2 weeks of payment'`, or `'between April 18, 2024 and April 25, 2024'`. This gives more flexibility for the contract author, and makes a relevant step towards full formalization. However, it will also add complexity to the processing, so the NL customizations will need to be controlled through the use of a restricted form of NL, known as a *Controlled Natural Language (CNL)*.

To summarize this goal more plainly, we are proposing a tool that allows the contract author (the user) to make controlled customizations to a NL contract template, which will result in predictable changes to a formal specification of the contract. To frame the problem in a more structured way: Given a contract template T written in NL, a manually created formal specification of this template S(T), and a customization C of the template,

we want to automatically generate the formal specification S(C) for the customization. This problem structure is shown in Fig. 1.1.

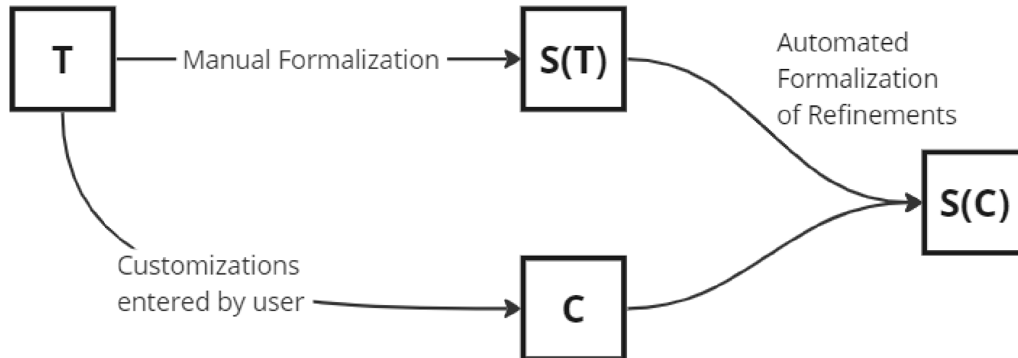


Figure 1.1: Diagram of our problem structure

We illustrate this process with our sample obligation and its corresponding SYMBOLEO specification, which might appear in a supply contract.

Template T: 'The seller shall deliver the goods to the buyer [REFINEMENT].'

- This is the NL contract template that is seen by the user, and they can fill in the REFINEMENT parameter with a CNL.

Formalization S(T): `Obligation(buyer, seller, true, Happens(evt_deliver_goods)`.

- This formalization is done manually and in advance by someone familiar with SYMBOLEO.

Customization C: 'The Seller shall deliver the goods to the Buyer *within 2 weeks of payment.*'

- The customization of the REFINEMENT is done by the user and accomplished through a guided and controlled process. For now we will assume that 'within 2 weeks of payment' is allowed in our CNL.

Formalization S(C): `Obligation(buyer, seller, true, WhappensBefore(evt_deliver_goods, Date.add(evt_payment, 2, weeks))`.

- This step is automated by the tool.

In this example, T is just a single obligation, but in practice (and throughout the rest of this thesis), it will represent a full NL contract template, consisting of multiple obligations and powers. Our task in this example is to use the context from the existing SYMBOLEO

template S(T) to determine how to properly formalize the customization `'within 2 weeks of payment'`. To automate this process, we must encode the fact that this customization corresponds to the operation of converting `Happens` to `WhappensBefore` in the SYMBOLEO specification. Likewise, any NL customization that we allow must have a reliable mapping to a SYMBOLEO operation. Once these mappings are defined, we will introduce a tool, SYMBOLEONLP, that will automate the mapping. Before building the tool, we must first determine which linguistic patterns we want to allow in our customizations, and this set of patterns will define our CNL.

1.3 Research Questions (RQs)

Our task involves mapping a SYMBOLEO specification S(T) to another SYMBOLEO specification S(C), which is achieved by performing a set of well-defined operations to S(T). Because the ultimate goal is the transformation of this specification, we first want to ask which operations we want our system to be able to handle. Since SYMBOLEO is a formal specification language with a finite set of concepts, there is a finite list of *possible* operations that could conceivably be done to a specification. It may not make sense for SYMBOLEONLP to support *all* of these, so we will use a set of guiding principles to decide which of these operations should be included in such a system.

Research Question RQ1

Which SYMBOLEO customization operations should be supported?

From the perspective of the contract author making use of SYMBOLEONLP, they would interact only with NL, and need not even know that SYMBOLEO exists. They only need to know that a valid specification is generated based on their NL input, and that this can be used for further smart contract applications. A hypothetical interface for the tool would present a NL contract template T, to which they make NL customizations C. The SYMBOLEO specification of the template S(T) has been loaded in manually by someone familiar with SYMBOLEO, and behind the scenes, the target SYMBOLEO contract S(C) will be automatically generated from these inputs. Therefore, once we've defined the desirable SYMBOLEO operations (RQ1), we need to match them to semantically equivalent NL structures. In our example, we had the phrase `'within 2 weeks of payment'` as the NL that corresponded to the SYMBOLEO `Happens` $\rightarrow$ `WhappensBefore` refinement operation. We will look more closely at the different NL structures that correspond to all of the operations defined in RQ1, which will give us a basis for our CNL.

Research Question RQ2

What NL structures correspond to our selected SYMBOLEO customization operations?

Since we are dealing with NL, there is necessarily going to be a great deal of complexity in attempting to process any user-generated input. The goal of RQ2 is to define a set of linguistic structures with *general* semantic equivalence to our SYMBOLEO operations, but ideally we would like these structures to be based on text from *real contracts*, rather than simplified examples. We will therefore treat the results of RQ2 as a set of *heuristics* for

NL refinement patterns that correspond to Symboleo operations. Using these heuristics as a starting point, we will analyze a dataset of real contracts to find the most common refinement patterns, providing an empirical basis for our CNL. We are looking specifically for patterns that commonly appear in contracts that correspond to our selected Symboleo operations.

Research Question RQ3

What linguistic refinement patterns found in real contracts can be used to construct a CNL?

The result of our first three RQs will be a CNL, consisting of an Extended Backus–Naur Form (EBNF) grammar specifying the allowed refinement patterns, as well as a set of mappings between these patterns and their corresponding Symboleo operations. This CNL will then be framed as requirements for our tool SymboleoNLP, which will provide a proof-of-concept for how this refinement process can be partially automated. SymboleoNLP will allow the user to construct realistic contracts in NL by customizing a template using the patterns defined in the CNL. It will then automatically generate a Symboleo specification of the customized contract, which can then be used in applications related to smart contract monitoring. We can therefore create two further research questions based on this description.

Research Question RQ4

How can we design a tool that enforces that the user enters input that adheres to our CNL?

Research Question RQ5

How can we design a tool that translates customizations expressed in the CNL into the proper Symboleo operations?

1.4 Contributions

The contributions of this thesis are as follows:

- A CNL that can be used to customize contracts in a flexible way, and which have a correspondence with logical operations on a Symboleo contract.
- A methodology and relevant artifacts for *constructing* such a CNL, so that the CNL can be iteratively improved upon.
- A manually-created dataset that labels refinements and their corresponding Symboleo operations. This may have value in related work in the growing Symboleo ecosystem.
- A configurable tool, SymboleoNLP, which demonstrates the automated formalization process against a set of realistic contracts. This tool consists of a fully-tested back-end, which includes a Python implementation of Symboleo, as well as a web-based user interface, allowing for easy interaction with the tool's core functionality.

We also note a first paper resulting from this thesis, mainly related to RQ1 to RQ3. I am the main author, with verification from my co-supervisors:

- **Meloche, R.**, Amyot, D., Mylopoulos, J. (2023) Towards Legal Contract Formalization with Controlled Natural Language Templates. In: *2023 31st IEEE International Requirements Engineering Conference (RE)*, Hannover, Germany, September 2023. IEEE CS, 317–322.

The CNL and the associated methodology will be of value to researchers focusing on legal contract formalization and related fields. The CNL construction is a step towards full formalization of legal contracts into Symboleo. As the Symboleo ecosystem expands, this CNL can be iterated upon by following the methodology described in this thesis. Furthermore, this methodology may also be re-purposed to create automated formalization processes for *other* formal specification languages, whether that be in the legal contract domain, government regulations, requirements engineering, or other domains that make use of normative language. The research will also be of interest to practitioners, particularly drafters and authors of legal contracts. The SymboleoNLP tool is designed for use by contract authors, and is designed so that authors do *not* require knowledge of Symboleo. Given a NL contract template, they will be able to easily make customizations in a more familiar CNL to suit specific client needs, and a resulting artifact will be a Symboleo specification of this contract. This can then be used with other applications in the Symboleo ecosystem related to the generation of smart contracts. This thesis offers a proof-of-concept of the tool's functionality and outlines ideas for developing it further into a fully functional application, which would include an empirical usability analysis.

1.5 Structure of Thesis

This work will proceed as follows: In Chapter 2, we will address some important preliminary concepts needed to fully frame the RQs and the development of SymboleoNLP. This includes details on the Symboleo language, relevant linguistics concepts, and NLP techniques that we may make use of later on. In Chapter 3, we will discuss prior work that is relevant to this thesis, including other contract specification languages, CNLs, and NLP techniques applied to the legal domain. Chapter 4 will involve answering our first three RQs, resulting in our CNL. We will also perform a detailed evaluation and analysis of our result and methodology. In Chapter 5, we will discuss how this CNL is applied to our tool, SymboleoNLP, and answer RQ4 and RQ5. We will step through some key design decisions involved with the tool, and walk through some concrete use cases to highlight the strengths and weaknesses. Chapter 6 will provide a more in-depth discussion of the entire work, reviewing our key results and points, identifying limitations and threats to validity, as well as brainstorming ideas for the future direction of this work. Finally, Chapter 7 will conclude the thesis.

Chapter 2

Preliminaries

We will now define and discuss important preliminary concepts, some of which have already been hinted at in the introduction. This will give us a clearer meaning of these concepts, which are used throughout this work, as well as some useful examples that we can refer back to. We will cover the relevant foundations of SYMBOLEO, as well as its syntax and semantics (Section 2.1). Since this work falls under the topic of NLP, we will discuss some important NLP principles and approaches that relate to our topic (Section 2.2). Finally, much of this work will be rooted in linguistic analysis, so we will highlight and provide examples of relevant linguistic concepts (Section 2.3).

2.1 Symboleo

2.1.1 Legal Contract Ontology

We have established that even simple NL terms can be ambiguous once we remove the assumption of common sense, and that this ambiguity is a major obstacle to formalization. We must therefore precisely define what is meant by the concepts commonly found in contracts (e.g., obligations, powers, assets). In addition to precise terminology, we are also interested in how the different contract terms *relate* to each other. For example, a contract obligation may refer to various events, which are in turn made up of assets and roles. Consider the obligation of a seller to deliver a quantity of meat to a buyer. This contains the event of *delivering* an item, and this event includes properties such as the delivering agent (seller), the delivery recipient (buyer), and the object to be delivered (meat). The contract may also contain a *power* that grants one party the ability to suspend one of the other obligations in the contract. The point is that the terms found in a contract may be related to each other in various ways, and identifying the structure of these relationships can guide us on how best to formalize a contract.

The needs of precise terminology and relatedness amongst terms are addressed by an *ontology*. An ontology gives precise meaning to the terms in a domain and captures structural relationships between those terms [35, 46, 91]. In addition to providing a more formal understanding of the domain, well-defined ontologies can also promote reuse across

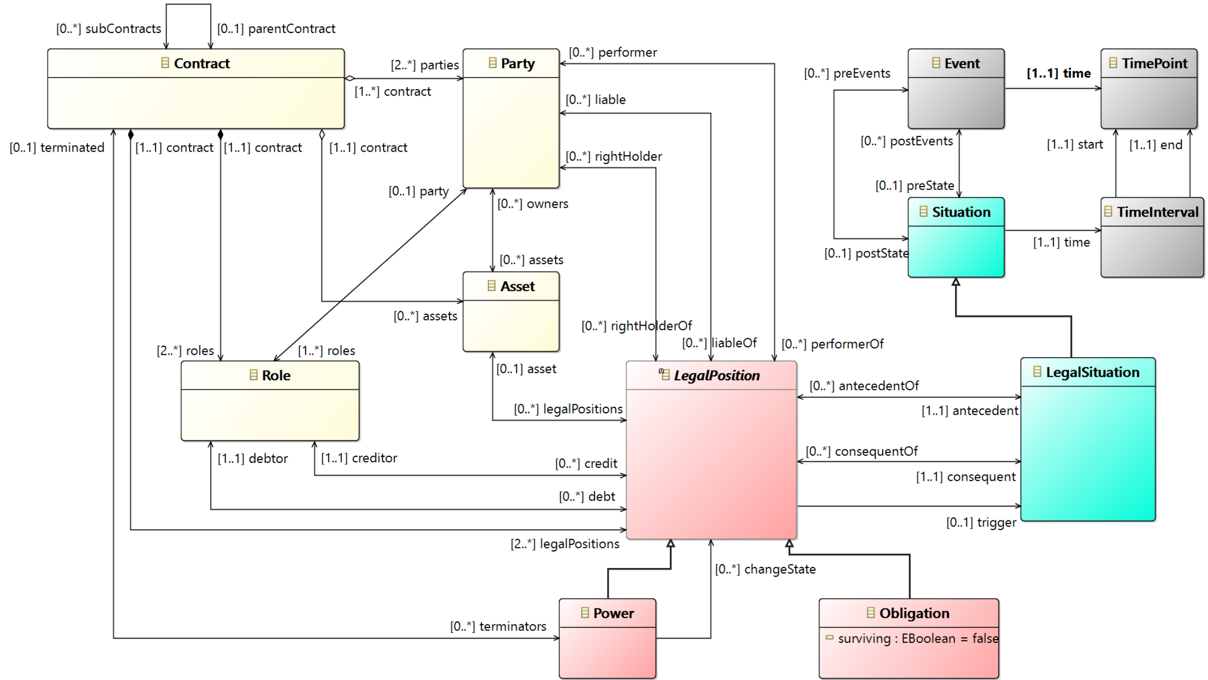


Figure 2.1: Symboleo ontology (from Parvizimosaed et al. [70])

multiple applications, by virtue of their various levels of granularity. A coarse-grained (foundational) ontology may define more abstract terms such as 'event', 'object', and 'relationship', whereas a fine-grained ontology defines more concrete concepts that pertain to a specific domain, such as that of legal contracts. A fine-grained domain ontology can extend a coarse-grained ontology, reusing the abstract concepts from the coarse-grained ontology as a foundation for new concepts. It is tempting to brush off abstractions like this as being unimportant, and indeed we might ask what an application *without* an ontology might look like compared to one that does use it. The reality, however, is that as soon as any concrete concepts are defined, we have made an *ontological commitment*. That is, we are making assumptions about the underlying abstract concepts. The opposite of an ontology is not *no ontology*, but rather *bad ontology* [41]. Conscious ontology design is important if we want our application to have sound foundational definitions as well as semantic interoperability within the greater legal technology ecosystem.

Symboleo is a specification language for legal contract monitoring and execution, and is based on an ontology created for the legal contract monitoring domain, which in turn was influenced by a set of more coarse-grained ontologies and foundations, including Hohfeld's legal conceptions [47], the Unified Foundational Ontology (UFO) [41], and UFO-Legal (UFO-L) [39]. We present the key concepts of the ontology [79, 80] (Fig. 2.1), as well as examples where relevant.

- **Contract:** A collection of obligations and powers between two or more roles, which are assigned to parties during execution, and are concerned with assets. It may also include subcontracts where a third party undertakes some obligations while the debtor retains the responsibility of the original contract.

- Example: A meat sale contract between a buyer and seller that outlines each of their obligations and powers.

- **Asset:** An owned item of value, which may be tangible or intangible.
  - Example: In our meat sale contract, the meat itself would be considered an asset. It may have additional properties, such as meat quality or amount.

- **Legal position:** Legal relationship between roles. Various ontologies use different collections of legal positions, such as permissions, obligations, and prohibitions. Symboleo focuses on obligations and powers, since most legal positions required for contract monitoring can be expressed as obligations or powers [80].

- **Obligation:** Legal duty of a debtor towards a creditor to bring about a certain legal situation (consequent) when another legal situation (antecedent) holds. In the case of unconditional obligations, the antecedent is always true. Obligations usually concern assets and are instantiated by a trigger condition.
  - Example: The buyer is obligated to pay the seller before the payment due date. Since there is no antecedent, this is considered an unconditional obligation.

- **Surviving obligation:** An obligation that remains in effect after the termination of a contract. Surviving obligations often take the form of *prohibitions*.
  - Example: The buyer is obligated to *not* disclose information about the contract for 6 months after the contract terminates. The seller has an equivalent obligation towards the buyer.

- **Legal situation:** State of affairs associated with a contract, obligation, or power. A situation occurs within a time interval T, but not in any proper sub-intervals of T.
  - Example: If the buyer fails the obligation of paying the seller before the payment date, then the obligation is considered to be *violated*.

- **Event:** A happening that occurs at a time instance, and cannot change.
  - Example: The action of delivering the meat is an event.

- **Power:** The right of a party to create, change, suspend, or terminate legal positions. A power is instantiated by a condition when the latter becomes true and has an antecedent (legal situation) that must be met for it to become in effect.
  - Example: If the buyer violates the payment obligation, then the seller has the power to *suspend* its own obligation of delivering the meat. This suspension might last until the payment has been made in full, in which case the seller has the additional power to *resume* the obligation.

- **Role:** A defined position in the contract that participates in obligations and powers.
  - Example: In our example, there are roles for a *buyer* and a *seller*.

- **Party:** A legal agent (person or entity) that is assigned to a role in a contract.
  - Example: Suppose the buyer on our meat sale contract is a specific grocer in Ottawa, Canada. This grocer would be the party to which the role of *buyer* is assigned.

In order to formalize NL contracts using this legal contract ontology, we require a formal specification language that contains the ontological concepts. Symboleo was developed for this purpose by studying a wide variety of business contracts and identifying the common concepts, such as obligations and powers (i.e., the norms). Each instance of a norm can be said to have a state, or a legal situation. For example, an obligation may be fulfilled, violated, suspended, etc. To transition between these states, Symboleo relies on the concept of an event. The underlying axioms that specify the semantics of Symboleo are therefore appropriately based on event calculus [78]. Symboleo also provides support for the dynamic aspects of contracts, such as subcontracting and post-agreement assignment of rights. For these reasons and others, Symboleo is a suitable candidate for the specification of smart legal contracts. We note that Symboleo is under active development, and that there may be future versions of the ontology and the Symboleo language that are different from the definitions and specifications provided in this work.

2.1.2 Sample Contract

Before formally describing the syntax and semantics of Symboleo, we present an example of a meat sale contract. The NL contract is shown in Table 2.1 and the corresponding Symboleo is shown in Listing 2.1.

This agreement is entered into effect as of March 1, 2024, between Beef Suppliers Ltd. as Seller with address 123 Main Street, and Greg's Grocer as Buyer with address 999 Central Ave.

Payment and Delivery
Seller shall sell an amount of 10kg meat with Prime quality to the Buyer.

Title in the Goods shall not pass on to the Buyer until payment of the amount owed has been made in full.

The Buyer shall pay $300 in CAD to the Seller before March 31, 2024.

The Seller shall deliver the Order in one delivery within 8 days of payment to the Buyer at its warehouse.

In the event of late payment of the amount owed, the Buyer shall pay a late fee equal to 5% of the amount owed, and the Seller may suspend performance of all of its obligations under the agreement until payment of amounts owed has been received in full.

Disclosure
Both Seller and Buyer must keep the contents of this contract confidential for 6 months following the termination of the contract.

Termination
Any delay in delivery of the goods will entitle the Buyer to terminate the Contract.

Table 2.1: Text from a sample Meat Sale contract [80]

Listing 2.1: SYMBOLEO specification of the meat sale contract

```
1  Domain meatSaleDomain // the Domain section of the contract
2    // participant models are defined using the Role type
3    Seller isA Role with returnAddress: String, name: String;
4    Buyer isA Role with warehouse: String, name: String;
5  
6    // Useful Enumerations are defined using the Enumeration type
7    Currency isAn Enumeration(CAD, USD, EUR);
8    MeatQuality isAn Enumeration(PRIME, AAA, AA, A);
9  
10   // The goods to be delivered are defined as an Asset
11   // Meat inherits the quantityKg property of PerishableGood
12   PerishableGood isAn Asset with quantityKg: Number;
13   Meat isA PerishableGood with quality: MeatQuality;
14 
15   // Event types
16   Deliver isAn Event with item: Meat, deliveryAddress: String, delDueDate: Date, from:
         Role, to: Role;
17 
18   Pay isAn Event with amount: Number, currency: Currency, from: Role, to: Role,
         payDueDate: Date;
19 
20   PayLate isAn Event with amount: Number, currency: Currency, from: Role, to: Role;
21 
22   Disclose isAn Event with agent: Role;
23 endDomain
24 
25 // Arguments are passed into the contract parameters to instantiate the contract
26 Contract MeatSale (
27   v_seller_name: String,
28   v_seller_address: String,
29   v_buyer_name: String,
30   v_buyer_address: String,
31   v_quantity_kg: Number,
32   v_quality: MeatQuality,
33   v_price: Number,
34   v_currency: Currency,
35   v_pay_due_date: Date,
36   v_start_date: Date,
37   v_delivery_days: Number,
38   v_interest_rate: Number)
39 
40 // Variables used throughout the contract are declared here
41 Declarations
42   seller: Seller with returnAddress := v_seller_address, name := v_seller_name;
43 
```

```
44    buyer: Buyer with warehouse := v_buyer_address, name := v_buyer_name;
45
46    goods: Meat with quantityKg := v_quantity_kg , quality := v_quality;
47
48    evt_pay: Pay with amount := v_price,
49      currency := v_currency,
50      from := buyer,
51      to := seller,
52      payDueDate := v_pay_due_date;
53
54    evt_deliver: Deliver with item := goods,
55      deliveryAddress := v_buyer_address,
56      delDueDate := Date.add(evt_pay.payDueDate, v_delivery_days, days),
57      from := seller,
58      to := buyer;
59
60    evt_pay_late: PayLate with amount := (1 + v_interest_rate / 100) * v_price, // Late
        penalty
61      currency := v_currency,
62      from := buyer,
63      to := seller;
64
65    evt_disclose_seller: Disclose with agent := seller;
66
67    evt_disclose_buyer: Disclose with agent := buyer;
68
69  // Safety conditions of the contract
70  Preconditions
71    IsOwner(goods, seller);
72
73  Postconditions
74    IsOwner(goods, buyer) and not(IsOwner(goods, seller));
75
76  Obligations
77    // This obligation requires seller to deliver before the due date
78    ob_delivery:
79      O(seller, buyer, true, WhappensBefore(evt_deliver, evt_deliver.delDueDate));
80
81    // Requires buyer to pay before the specified due date
82    ob_payment:
83      O(buyer, seller, true, WhappensBefore(evt_pay, evt_pay.payDueDate));
84
85    // Requires the buyer to pay the late fee
86    ob_late_payment:
87      Happens(Violated(obligations.ob_payment)) -> O(buyer, seller, true, Happens(
          evt_pay_late));
88
89  // These are the obligations not to disclose confidential information
90  Surviving Obligations
91    so_disclosure_seller:
92    O(seller, buyer, true, not WhappensBefore(evt_disclose_seller, Date.add(Terminated(
        self), 6, months)));
93
94    so_disclosure_buyer:
95    O(buyer, seller, true, not WhappensBefore(evt_disclose_buyer, Date.add(Terminated(self
        ), 6, months)));
96
97  Powers
98    // If payment is violated then seller can suspend the delivery obligation
99    pow_suspend_delivery:
100     Happens(Violated(obligations.ob_payment)) ->
101     P(seller, buyer, true, Suspended(obligations.ob_delivery));
102
103   // If penalty is paid then buyer can resume the delivery obligation
104   pow_resume_delivery:
105     HappensWithin(evt_pay_late, Suspension(obligations.ob_delivery)) ->
106     P(buyer, seller, true, Resumed(obligations.ob_delivery));
107
108   // If delivery is violated then buyer can terminate the contract
109   pow_terminate_contract:
```

```
110     Happens(Violated(obligations.ob_delivery)) -> P(buyer, seller, true, Terminated(self
          ));
111
112   Constraints
113     not(IsEqual(buyer, seller));
114
115   endContract
```

2.1.3 Main Components

Domain Model

A Symboleo contract has two main components: the domain model and the contract specification. The domain model contains templates for roles, assets, and events, which are used as building blocks for the obligations and powers defined in the contract specification. These three different domain classes are characterized by their type (superclasses Asset/Role/Event from the ontology), a named identifier, and a collection of domain properties. Each domain property consists of a key and a type. To be used in the contract specification, these domain classes are instantiated as 'declarations'. For example, the 'Seller' is a Role that consists of a returnAddress and a name, both of which are strings. When the contract itself is instantiated, we declare a specific instance of the seller, using the arguments for the v_seller_address and v_seller_name parameters. Once declared, the 'seller' instance can be used in other parts of the contract.

The assets and events work similarly. We have a 'PerishableGood' asset, which has a numerical property of quantityKg, referring to the quantity of the good in kilograms. Another asset is 'Meat', which inherits from the 'PerishableGood' asset, and in doing so, inherits the quantityKg property. It also has its own property of quality, which specifies the quality of the meat. This property has the type 'MeatQuality', which is defined as an enumeration type in the domain model as well. An instance of the 'Meat' asset identified as 'goods' is declared in the *Declarations* section.

Finally, an example of a domain event is the 'Deliver' event, which has a property called item of type 'Meat' (*what* is begin delivered), a string property called deliveryAddress (*where* the item will be delivered), and a date property called delDueDate (*when* the delivery must take place by). A delivery event therefore consists of a meat item to be delivered, an address to which the meat is delivered, a due date before which the delivery may need to be completed, and the to/from role assignments. In our sample contract, these properties are once again filled with values from the arguments passed in to the contract. Below is a sample instantiation of the contract with arguments based on the NL contract in Table 2.1:

```
my_contract = MeatSale(
    v_seller_name = "Beef Suppliers Ltd.",
    v_seller_address = "123 Main Street",
    v_buyer_name = "Greg's Grocer",
    v_buyer_address = "999 Central Ave.",
    v_quantity_kg = 10,
```

```
        v_quality = MeatQuality(PRIME),
        v_price = 300,
        v_currency = Currency(CAD),
        v_pay_due_date = Date(2024/03/31),
        v_start_date = Date(2024/03/01),
        v_delivery_days = 8,
        v_interest_rate = 5
    )
```

Contract Specification

Once these arguments are passed in, further declarations are assembled in the 'Declarations' section of the contract specification. They are then used to construct the powers and obligations, which are formatted as follows:

Obligation `O-id: [trigger ->] O(debtor, creditor, antecedent, consequent)`

Power `P-id: [trigger ->] P(debtor, creditor, antecedent, consequent)`

The SYMBOLEO language permits both `O(...)` and `Obligation(...)` as alternative syntaxes for obligations, as well as `P(...)` and `Power(...)` for powers.

In an obligation, the debtor is the party that must fulfill the obligation, and the creditor is the party that benefits from the obligation. In a power, the debtor is the party that has the right to exercise the power, and the creditor is the party that is liable for the power. We can characterize an obligation as a legal duty of a debtor towards a creditor to bring about a certain legal situation (consequent) when another legal situation (antecedent) holds. In the case of a power, a true antecedent grants the debtor the right to make the consequent true by exerting their power. The consequent of a power will always include a reference to another obligation, power, or the full contract itself. For example, a power may allow a party to resume a suspended obligation, suspend an existing power, or terminate the entire contract.

The trigger is another propositional component that *creates* a new instance of an obligation or a power. Before considering the truth value of a norm's antecedent, the norm must first be created, which is the responsibility of the trigger. The distinction between the trigger and the antecedent will be clarified when we discuss the precise semantics of SYMBOLEO, but from a higher-level semantic perspective, both propositions act as conditions for the consequent of a norm. We now look at two examples from our sample contract in more detail.

First, consider the *payment* obligation, which specifies that the buyer must pay before the due date: `ob_payment: O(buyer, seller, true, WhappensBefore(evt_pay, evt_pay.payDueDate))`. There is no trigger for this obligation, meaning it is created as soon as the contract begins. Furthermore, the antecedent is `true`, meaning the obligation is also in effect as soon as the contract begins. An obligation with a true antecedent is therefore known as an unconditional obligation. The buyer (filling the debtor role) is

unconditionally obligated to make the payment to the seller (who fills the creditor role) before the specified due date. If the payment is not made before the due date, then this obligation will be considered *violated*.

Next consider the *pow_suspend_delivery* power: `Happens(Violated(obligations.ob_payment)) -> P(seller, buyer, true, Suspended(obligations.ob_delivery))`. This power is created in the case of a violation of our previous payment obligation (i.e., the buyer has failed to pay by the due date). If that obligation is fulfilled, then we need not concern ourselves with this power. However, if payment does not happen on time, then this power is *triggered*, meaning it is created. Since there is no antecedent, it is immediately in effect as well. The power specifies that the debtor (the seller) can suspend the delivery obligation. In other words, the seller no longer needs to deliver the goods to the buyer if the buyer has not paid on time.

While this work will mainly be concerned with the norms and the declarations from which they are constructed, there are a few other concepts that are part of the contract specification that we will mention:

- **Preconditions and Postconditions:** These are propositions that must hold before and after the execution of the contract, respectively. In our sample contract, these deal with the ownership of the goods.
- **Constraints:** Constraints are propositions that help ensure the logical flow of a contract. We can think of these as codifying assumptions about contracts that are often implicit. Safety constraints ensure that bad things do not happen during execution, such as one party being assigned to both the buyer and seller roles, Liveness constraints ensure that contract execution terminates in a bounded amount of time.

2.1.4 Syntax of Symboleo

The legal contract ontology discussed in the previous section provides relatively precise definitions of the relevant terminology, but they are still expressed in NL and therefore not yet precise enough for a machine level. To this end, a formal syntax for SYMBOLEO is specified in an EBNF grammar, as well as an XText specification [1]. It is this syntax to which we must adhere in order to produce valid SYMBOLEO contracts. We emphasize once more that SYMBOLEO is under active development, and that there may be future versions of SYMBOLEO that are different from this specification. For this thesis, we are fixing our version of the SYMBOLEO syntax as the one referenced in the preceding footnote[1].

One of the most important concepts in the syntax which we've alluded to several times is the *Proposition*. The trigger, the antecedent, and an obligation's consequent all take the form of a proposition. Generally speaking, this is an expression that can evaluate to true or false. There are a variety of formats a proposition can take, including inequalities and predicate statements, and it supports the logical operations of conjunction, disjunction,

[1] `https://github.com/Smart-Contract-Modelling-uOttawa/Symboleo-IDE/blob/master/ca.uottawa.csmlab.symboleo/src/ca/uottawa/csmlab/symboleo/Symboleo.xtext`

and negation. We present a simplified form of a sample of the XText syntax in Listing 2.2 to better illustrate this concept:

Listing 2.2: Sample of the SYMBOLEO syntax

```
Obligation:
    name=ID ':' (trigger=Proposition '->')? ('O' | 'Obligation')
       '(' debtor=VariableDotExpression ',' creditor=
       VariableDotExpression ',' antecedent=Proposition ','
       consequent=Proposition ')';

Proposition: POr;

POr returns Proposition:
    PAnd ({POr.left=current} 'or' right=PAnd)*;

PAnd returns Proposition:
    PEquality ({PAnd.left=current} 'and' right=PEquality)*;

PEquality returns Proposition:
    PComparison ({PEquality.left=current} op=('==' | '!=') right=
       PComparison)*;

PComparison returns Proposition:
    PAtomicExpression ({PComparison.left=current} op=('>=' | '<='
       | '>' | '<')  right=PAtomicExpression)*;

PAtomicExpression returns Proposition:
  {PAtomRecursive} '(' inner=Proposition ')' |
  {NegatedPAtom} 'not' negated=PAtomicExpression |
  {PAtomPredicate} predicateFunction=PredicateFunction |
  {PAtomFunction} function=OtherFunction |
  {PAtomEnum} enumeration=[Enumeration] '(' enumItem=[EnumItem] '
     )' |
  {PAtomVariable} variable=VariableDotExpression |
  {PAtomPredicateTrueLiteral} value='true' |
  {PAtomPredicateFalseLiteral} value='false' |
  {PAtomDoubleLiteral} value=Double |
  {PAtomIntLiteral} value=INT |
  {PAtomStringLiteral} value=STRING |
  {PAtomDateLiteral} value= Date
  ;

PredicateFunction:
    {PredicateFunctionHappens} name='Happens' '('event=Event')' |
    {PredicateFunctionWHappensBefore} name='WhappensBefore' '('
       event=Event ',' point=Point ')' |
    {PredicateFunctionSHappensBefore} name='ShappensBefore' '('
       event=Event ',' point=Point ')' |
    {PredicateFunctionHappensWithin} name='HappensWithin' '('
```

```
    event=Event ',' interval=Interval ')' |
{PredicateFunctionWHappensBeforeEvent} name='WhappensBeforeE'
     '(' event1=Event ',' event2=Event ')' |
{PredicateFunctionSHappensBeforeEvent} name='ShappensBeforeE'
     '(' event1=Event ',' event2=Event ')' |
{PredicateFunctionHappensAfter} name='HappensAfter' '(' event
    =Event ',' point=Point ')' |
{PredicateFunctionOccurs} name='Occurs' '(' situation=
    Situation ',' interval=Interval ')' ;
```

The proposition can consist of a pair of disjuncts, each of which can consist of a pair of conjuncts, which can support concepts of equality, comparison, or a PAtomicExpression. The PAtomicExpression, in turn, can take the form of a boolean value, a negation followed by *another* PAtomicExpression, or a PredicateFunction. This shows how SYMBOLEO supports basic logical operations such as conjunction, disjunction, and negation. The PredicateFunction is the most relevant form of proposition to us, as these are prominent in the norms found in real contracts. There are various types of predicate functions, adapted from event calculus and shown alongside a basic semantic interpretation in Table 2.2. The consequent of a power is a more narrow type of proposition that can only include references to existing obligations, powers, or the contract itself.

Predicate	**Semantics**
`Happens(e)`	event `e` happens
`HappensAfter(e, t)`	event `e` happens after timepoint `t`
`WhappensBefore(e1, t)`	event `e1` happens before timepoint `t`, and either timepoint `t` passes or not
`ShappensBefore(e1, t)`	event `e1` happens before timepoint `t` and timepoint `t` passes eventually
`HappensWithin(e, i)`	event `e` happens within an interval of time `i`
`WhappensBeforeEvent(e1, e2)`	event `e1` happens before event `e2`, and either `e2` happens or not
`ShappensBeforeEvent(e1, e2)`	event `e1` happens before event `e2`, and `e2` happens eventually
`Occurs(s, i)`	Situation `s` holds over an interval of time `i`

Table 2.2: Predicates of Symboleo

Each of these predicates has a set of arguments, which are the building blocks of the predicates, which in turn are the building blocks of the norms. These arguments are precisely defined in the grammar, but we will provide a brief semantic overview of each.

- **Timepoint (or Point):** a point in time. It can have many representations, including a simple date object, the starting point of an event, or a function that specifies a time relative to some anchor timepoint (e.g., 3 weeks after Event X)
- **Event:** A happening that occurs at a timepoint, and cannot change. Specific examples of events include pre-defined contract events (contract activation, termination),

events related to the contract norms (obligation fulfillment, power suspension), or declarations of events defined in the domain model (Delivered, Paid).

- **Situation:** A states of affairs that persists over an interval of time. In the current version of the SYMBOLEO syntax, a situation refers specifically to the state of the contract, an obligation, or a power. An example of a situation could be `Fulfilled(my_obligation)`, which means that *my_obligation* is in the fulfilled state.

- **Interval:** An interval can be defined by a starting timepoint and an ending timepoint, or it can be represented by a situation, as described above.

There are many syntactic relationships between these concepts, which will have important implications when we consider the various ways they can be expressed in NL. We present a diagram in Fig. 2.2, which represents a subset of these predicates and their arguments to illustrate these relationships. The solid lines show membership and the dotted lines show the possible concrete forms that a concept can take. For example, the `WhappensBefore` predicate requires an event and a point. An event can take the form of a Contract Event, Power Event, Obligation Event, or Variable Event. An Obligation Event, for example, refers to an event that specifically happens to an instance of an obligation (e.g., the obligation becomes fulfilled). Similar definitions exist for the Contract Event and Power Event. A Variable Event can refer to an event declaration based off a Domain Model Event (e.g., evt_deliver). A Point consists of a PointExpression object, which can take the form of a Point Function or a Point Atom. A Point Atom, in turn, can take the form of a Contract Event or a Variable Dot Expression (VDE), which may be a reference to a specific date variable.

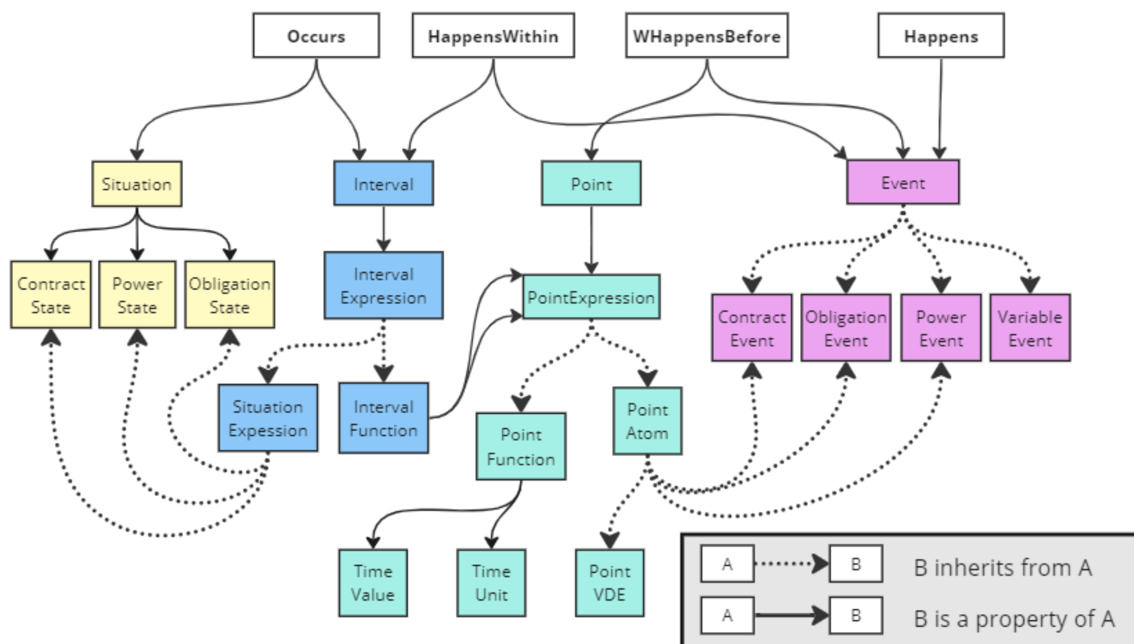


Figure 2.2: Diagram showing relationships between predicates and their arguments

We must now make an important distinction between an event and a situation. We mentioned that a *Situation* in SYMBOLEO specifically applies to obligations, powers, and contracts. This raises the question of how SYMBOLEO can represent *other* types of situations that are expressed in NL. For example, consider the following NL obligation: 'Buyer must keep the contents of this contract confidential'. The key verb phrase here refers to 'keeping the contents confidential'. In NL, we might consider this to be a *situation* rather than an *event*. Events must take place at a specific point in time, whereas this particular statement suggests a *duration*. Since the current SYMBOLEO syntax cannot represent these NL situations *directly*, we must use a different approach. In general, for a given situation that is part of an obligation – such as in our example – we can re-frame the obligation as a *negated event*. The situation of 'keeping contents confidential' is synonymous with 'Not disclosing the contents'. This is what is meant by a *negated event*. If we think of the obligation as an event, we can now represent it in SYMBOLEO, as follows: `O(buyer, seller, true, not Happens(evt_disclose))`.

We assume that there exists an event in the domain model called Disclose as well as a corresponding declaration called evt_disclose. As we process NL contracts in this work, we emphasize that we do not want to change the actual wording on a given NL contract from describing a situation to describing a negated event. We simply note that this conceptual re-framing must be done when manually converting a NL contract to a SYMBOLEO specification. We will see further examples of this later on. Going forward, this means that we will rarely see the `Occurs` predicate being used, as it is specifically for SYMBOLEO situations, which are limited to states of obligations, powers, and the contract itself. A practical drawback of this reality from a contract monitoring perspective is that it is difficult to monitor durations and *non-occurrences* of events, since there must be some sort of constant polling to verify the target condition.

2.1.5 Semantics of Symboleo

The semantics of SYMBOLEO are specified in a series of axioms influenced by event calculus and are equivalently represented in a state chart diagram. Some illustrative examples of these axioms are presented below, and the state chart diagram representing this subset of axioms is shown in Fig. 2.3.

Axiom 1 (Create a conditional obligation): Given a conditional triggered obligation o of contract c, if o is triggered while c is in effect, then o is created.

Axiom 3 (Convert a created obligation to an effective one): Given an obligation o of contract c, if an event e fulfills the antecedent of o while o is created and c is in effect, then o becomes effective.

Axiom 6 (Fulfill an obligation): If the consequent of an effective obligation o becomes true, then o is fulfilled.

Axiom 10 (Suspend an obligation by a power): Given an obligation o and a power p, if the consequent of p implies that o is suspended and the event happens while p is in effect, then o is suspended.

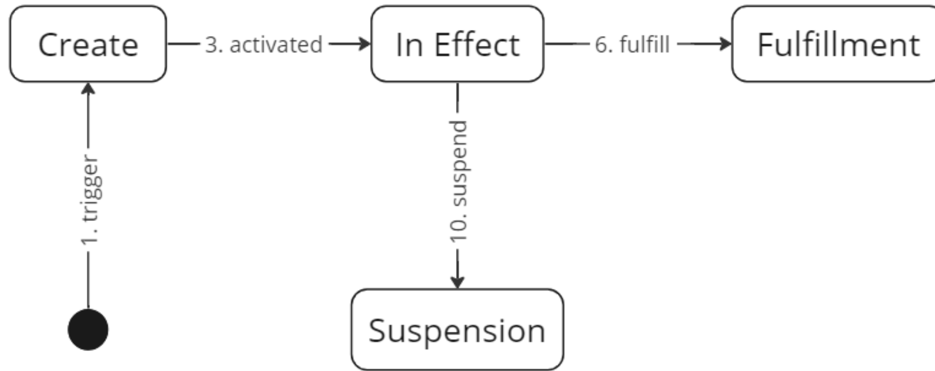


Figure 2.3: Simple state diagram based on a small subset of the axioms

For example, we say that a trigger instantiates an obligation. This is equivalent to saying that a triggering event transitions an obligation to the *created* state. When the antecedent becomes true, the obligation is *in effect*. The obligation can be suspended by a power, or it can be fulfilled if the consequent becomes true. The full set of axioms yields a much more sophisticated series of state diagrams for obligations, powers, and the contract itself. These are presented in Fig. 2.4, which represents the full semantics of Symboleo contracts and legal positions.

At this point we must address the semantic difference between a trigger and an antecedent. We've previously mentioned that they are both, in a sense, conditional. According to the semantics however, the trigger *creates* a norm instance, and the antecedent puts the norm into *effect*. An example can help illustrate the difference. Suppose we have a rental contract with an obligation that the renter must pay rent to the landlord: `pay_rent: O(renter, landlord, true, Happens(evt_pay_rent))`. This is a straightforward unconditional obligation. Now suppose that this is a *recurring* obligation, and the renter must make this payment each month. This means that we need a new *instance* of this obligation each month, and this would be captured in a trigger statement. A new 'pay_rent' obligation must be instantiated each month through the use of a trigger. While this illustrates the distinction between a trigger and antecedent, we note that Symboleo currently does not support the notion of recurrence, but this may be incorporated in future releases. The antecedent is responsible for transitioning an obligation to being *in effect*. A `true` antecedent is automatically in effect upon creation. Suppose our rental obligation also included the condition that the payment is only to be made if the apartment building has been recently cleaned. This could be an example of an antecedent. The rental payment obligation could therefore *exist*, but not be *in effect* until the landlord has cleaned the building. This might be represented as follows: `pay_rent: Happens(evt_month_begins) -> O(renter, landlord, Happens(evt_clean_building), Happens(evt_pay_rent))`.

We've now described the formal syntax and semantics of Symboleo. If a NL contract was specified purely in the semantics we just described (for example, by using terms such as 'instantiate the power', 'fulfill the obligation', etc.), it would be almost trivial to transform such a document into a Symboleo specification. However, we cannot assume that drafters of legal contracts will be familiar with the syntax and semantics of

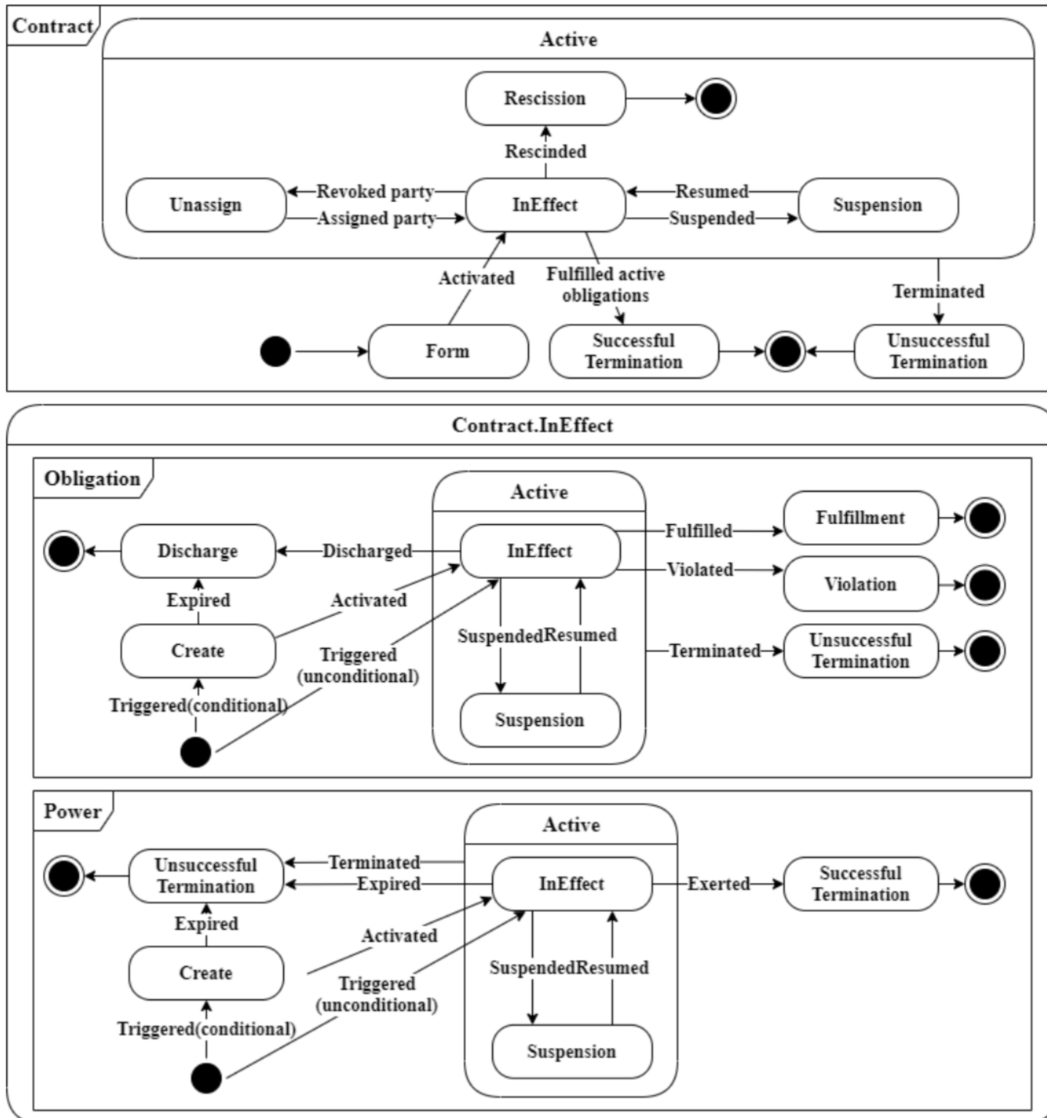


Figure 2.4: Full state charts for contract, obligation, and power

Symboleo. They will understandably want to use more expressive and flexible NL. The challenge will be to find correspondences between NL semantics that appear in contracts and the semantics of Symboleo, which can facilitate the automated formalization of contract refinements. The development of tools to implement this formalization is one of the next main objectives for the development of Symboleo, and this is precisely where the work presented in this thesis fits in.

2.2 Natural Language Processing

In this section, we will provide a brief introduction to aspects of NLP that are relevant to our work.

2.2.1 Formalization

Any task that involves NL as an input or an output can fall under the umbrella of NLP. Examples include sentiment analysis, conversational chatbots, and information extraction. Aspects of formalization are also closely linked with certain NLP tasks, particularly information extraction. The goal of information extraction is to extract a salient span of text from a longer piece of text. For example, a company may want to extract all mentions of their product from a collection of tweets for marketing purposes. A law firm may want to extract exceptions to a regulation from a legal document to find out if a client broke the law. *Formalizing* a legal document involves finding relevant terms from the document that fit into some predefined template (the formal specification language) that represents the concepts that are relevant to us. The structure of these templates will depend on our use case.

Suppose we are interested in extracting legal obligations from a contract. Some use cases may only require that we extract the spans of text that include an obligation. This might be a use case for a basic contract review application, where a legal expert needs to efficiently find any potential obligations for their client. Other applications might require that the *parties* relevant to the obligation be also identified (e.g., creditor and debtor) in addition to the span of text. For an application that monitors smart contracts, we need to extract obligations in the form of Symboleo obligations, meaning we need to identify the debtor, creditor, trigger, antecedent, and consequent. Still other applications may require even *more* fine-grained information.

The point is that the tasks of information extraction and formalization are closely related. In both tasks, we are extracting textual information into a predefined representation. Just as any formal representation makes an ontological commitment, any information extraction task makes a *representational commitment* (i.e., a representational structure is implied). Some tasks may simply use a span of text to represent a concept, while others require more fine-grained properties. The designation of *formal* representation is conventionally reserved for these more detailed representations that have an unambiguous interpretation. This insight is validated by existing research, where the same NLP techniques are used for tasks labeled as 'information extraction' and tasks labeled as 'formalization' [36, 40, 53], which we will discuss in more detail in the next chapter. Framing the formalization task in this way allows us to explore a variety of NLP techniques that may help solve our problem.

2.2.2 Machine Learning (ML) Approaches

While the techniques used for solving NLP tasks such as formalization are diverse and growing, we will discuss two broad tendencies in the field related to the present work: Supervised machine-learning (ML) based approaches and linguistic-based approaches. ML-based approaches involve identifying patterns in existing data that can be used to make predictions on new data.

Suppose we want to create a model that predicts if an email should be labeled as spam. We would feed our model thousands of emails that are labeled as being spam or not

spam (sometimes called 'ham'). The fact that these emails are labeled is what makes this *supervised.* These labels tell the model which emails are definitively spam or ham so that it can learn to classify future emails correctly. The model might look for features that are present in many of the spam messages and absent from many of the ham messages. When a new email comes in, the model makes a prediction based on the configuration of these features. An important aspect of many traditional machine learning methods is feature engineering. This is where we decide which features are important for the model to consider. For classifying spam emails, this might include the average word length, the percentage of punctuation characters, the number of prepositions, etc. The role of a feature engineer is to decide which of these features are relevant to the task at hand.

Over the past decade, a key technique that has emerged in the field of NLP is *word embeddings* [63]. a word embedding is a vector representation of a word in some n-dimensional space. By representing words as vectors, mathematical operations can be performed on them, and embeddings are often designed so that similar words are geometrically close in the vector space. For example, the normalized distance between similar words such as 'water' and 'ice' may be 0.8, and the distance between dissimilar words such as 'water' and 'unicycle' might be closer to 0.1. The words for a given piece of text can be converted to these vector embeddings, which can then be used as features for a machine learning model. This is by no means a perfect representation of words, and it can perpetuate biases that are found in the corpus from which it is built [11, 38, 98].

Traditional ML classifiers include decision trees, support vector machines, naive Bayes, and logistic regression. These are often contrasted with 'deep learning' approaches, which are based on artificial neural networks, and involve finding some combination of input weights in order to minimize a loss function. From a NLP perspective, an important consideration missing from many these approaches is the sequential nature of language. In most NLP applications, word order will be important, and this reality is not handled by some of these traditional classifiers. Therefore a sentence such as 'the pizza was small and not very tasty' would be equivalent to 'the pizza was not small and very tasty', which has a very different meaning. This drawback has been addressed in part by more complex variations on deep learning, such as Recurrent Neural Networks (RNNs) and long-short-term memory Long-Short-Term Memory (LSTM) networks. These models have mechanisms for handling sequential data, but performance can degrade if they need to keep track of longer sequences of text [7, 71, 92].

More recently, we have seen the growth of transformer-based models, such as Bidirectional Encoder Representations from Transformers (BERT), which address this issue by using an attention mechanism that allows the model to focus on the relevant areas of a text, regardless of how long it is. These models have received a lot of attention in recent years due to strong performance on certain language tasks [22], and they are being explored in many areas, including the legal domain [17, 18]. Transformer-based approaches typically involve building a sophisticated language model by pre-training on a huge corpus of NL documents, and then transferring the knowledge encoded in that model to a specific type of language problem, such as text classification or information extraction, through a process called *fine-tuning.*

Despite the strong performance of these models, there are some notable drawbacks. We have already mentioned how bias can be transferred from a training corpus into the

language model. Another drawback is the lack of explainability [34, 50]. Results achieved from these models are largely rooted in the statistics-based model construction from a large corpus of NL text. Such a language model is going to make probabilistic NL predictions based on data that it has seen in the past, rather than explicit logical reasoning about the linguistics in the input. Furthermore, any supervised machine learning model requires labeled data to learn. In many cases, these labels must be obtained by using human experts, which can be a tedious, time-consuming, and expensive process. These realities may make a supervised ML approach for certain NLP tasks prohibitive, particularly in the legal domain. Fortunately, there are cases where we can leverage the power of a model that has already been trained on a large labelled corpus, and apply it to our unique scenario.

2.2.3 Linguistic Approaches

Linguistic-based approaches to NLP tasks involve analyzing the various linguistic structures of the text we are processing. A common initial step in this type of approach is to run our input text through a *standard NLP pipeline*, which involves adding important linguistic context to the tokens that make up our input text. This context might include parts of speech (verb, noun, adverb, etc.), word lemmas, syntactic dependencies between the words, and more sophisticated semantic structures, such as noun phrases or verb phrases. An example of this is shown in Fig. 2.5 and Fig. 2.6.

Sentence: The buyer must pay $100 to the seller

i	TEXT	POS	TAG	DEP	LEMMA	HEAD	ENT
0	The	DET	DT	det	the	buyer	
1	buyer	NOUN	NN	nsubj	buyer	pay	
2	must	AUX	MD	aux	must	pay	
3	pay	VERB	VB	ROOT	pay	pay	
4	$	SYM	$	nmod	$	100	
5	100	NUM	CD	dobj	100	pay	MONEY
6	to	ADP	IN	prep	to	pay	
7	the	DET	DT	det	the	seller	
8	seller	NOUN	NN	pobj	seller	to	

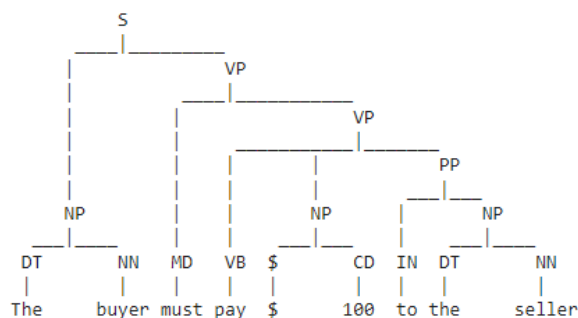


Figure 2.5: Linguistic information extracted from the sentence 'the buyer must pay $100 to the seller'

Algorithms that make decisions based on linguistic characteristics of the text can then be applied. Let us return again to our task of checking if a sentence contains a legal

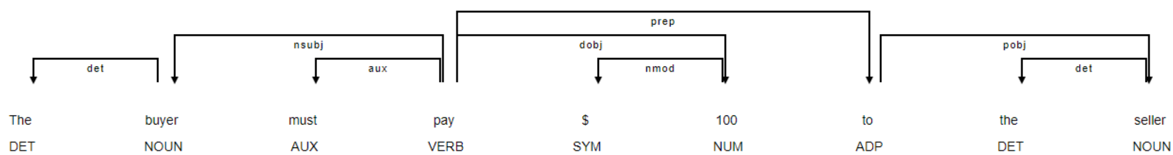


Figure 2.6: Dependencies extracted from the sentence 'the buyer must pay $100 to the seller'

obligation. A very simple heuristic may be to check for the presence of a modal verb, such as 'shall' or 'must'. This is sometimes called a *trigger word/phrase* approach. If the trigger phrase is found, then we claim to identify an obligation. Alternatively, we may want to enforce a stricter linguistic structure to characterize an obligation. We might decide that an obligation must contain a subject followed by a modal verb, followed by an action verb, and then a direct object. There may also be optional adverbs or prepositional phrases. We could capture this specification as a grammar in EBNF form in Listing 2.3.

Listing 2.3: Grammar for specifying an obligation

```
<OBLIGATION> ::= <SUBJ> <MODAL_VERB> <ACTION_VERB> <DOBJ> [<
   ADVERB>] [<PREP_PHRASE>]*
<SUBJ> ::= (noun_phrase)
<MODAL_VERB> ::= "must" | "shall"
<ACTION_VERB> ::= (verb)
<DOBJ> ::= (noun_phrase)
<ADVERB> ::= (adverb)
<PREP_PHRASE> ::= <PREPOSITION> <POBJ>
<PREPOSTION> ::= "with" | "from" | "to" | ...
<POBJ>::= (noun_phrase)
```

Suppose we pass in the input text 'the buyer must pay $100 to the seller'. A successful NLP pipeline might identify 'the buyer' as the subject, 'pay' as an action verb, '$100' as the direct object and so on. The pattern of the input text would match the grammatical pattern we defined for an obligation and the text would be classified as such. More sophisticated rules involving dependencies, verb lemmas, and sentence constituents can be built around the more complex linguistic concepts that are extracted from the text.

The immediate benefit of a linguistic-based approach over a machine-learning approach is explainability. If a sentence is classified as an obligation, then we know precisely *why* that decision was made. Furthermore, this approach does not directly require a labeled dataset. Despite these benefits, results achieved with a linguistic-based approach tend not to scale as well to unseen data or new domains [18, 51, 66]. It is important to emphasize that these two approaches are by no means mutually exclusive, and are often *combined* in many NLP tasks. A machine learning model may use linguistic properties as part of its engineered feature set. The extraction of linguistic properties (e.g., parts of speech, dependencies) used in the linguistic-based approach is often performed by a model built from a supervised ML approach. The English language is too complex and varied to simply use a dictionary lookup to determine the linguistic character of every word that is

encountered in a text. Instead, NLP pipelines make use of models that have been trained on large volumes of text that have been meticulously labeled by human experts [10]. A verb may be classified as a verb not because a dictionary lists it as a verb, but rather because the classifier picked up a pattern in the labeled corpus that suggests with high probability that it is a verb. These large labelled datasets allow for the creation of models that are able to automatically annotate text with linguistic information with a high success rate. We can then apply this model to a different problem, such as the legal domain. This is precisely what was being done in Fig. 2.5. The linguistic information was extracted using a ML-based model, allowing us to then write rules that are based on these linguistic characteristics. Transferring a model in this way is a very useful technique, though it is not always possible, and due to the potential of carrying over bias from the dataset, it must be done with care.

2.2.4 Controlled Natural Language

The techniques mentioned above are valid in many NLP tasks, particularly in cases where we need to work with unrestricted textual input (such as existing legal documents, movie reviews, or tweets), which may be filled with ambiguities and grammatical errors. The NLP techniques can impose some structure on the text, but the complexity of unrestricted NL makes this difficult and, in many cases, too probabilistic to be practical. An important variation on these tasks, however, involves processing text that is *not yet generated*, where we have some degree of control over the structure of the text that is being processed. For example, we could construct a vocabulary that disallows certain ambiguous words, enforces proper grammar, and restricts the length of the sentences. This is an example of a *Controlled Natural Language* (CNL), and it can make the processing of the text much more (but not completely) predictable. The downside, however, is that a restricted language is less expressive and may feel awkward to someone using the language. For example, if contract authors were forced to write directly in Symboleo, processing would be trivial and our problem statement would disappear. However, this language (like many formal specification languages) is unnatural and the authors would need extensive training on how to use it properly. This is precisely the dilemma in which many CNLs arise. Properly designed, a CNL is understandable by the targeted users, but is restricted in syntax and/or semantics to the point where it can be unambiguously represented, and can therefore be interpreted by a machine. We can look to various definitions of CNLs in the existing literature, many of which are well-aligned with each other.

- CNLs are a restricted subsets of natural language that have been designed with a specific aim, restricted in vocabulary, syntax, and semantics, to enable the language to be learned, translated, analysed, and generated (Pace and Rosner [67]).
- CNLs are engineered languages that use a selected vocabulary, morphological forms, grammatical constructions, semantic interpretations, and pragmatics that are found in natural language (Wyner et al. [95]).
- A CNL is a well-defined subset of natural language that has been assigned an unambiguous formal semantics (Hoefler and Bünzli [45]).

- CNLs are subsets of human languages, such as English or Chinese, and use restricted grammar rules and vocabularies (typically between 800 and 1,000 words) to reduce or eliminate ambiguity and complexity (Finnegan [26]).
- A CNL is a subset of standard English with a domain-specific vocabulary and a restricted grammar (Fuchs et al. [32]).
- A CNL is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, and/or semantics, while preserving most of its natural properties (Kuhn [55]).

Some common elements in these definitions include the idea that a CNL is a *subset* of NL, and that it reduces ambiguity by restricting the language in some way. It is worth noting that the idea of a *subset* is specifically omitted from Kuhn's definition, since certain CNLs actually *add* features to the NL, often for the purpose of clarification. An example could be using colours to indicate certain types of text or a new use for parentheses. This definition of a CNL is made in the context of a rigorous survey of many English-based CNLs that go as far back as the 1930s, with Basic English, which was designed with the goal of making it easier for non-native English speakers to learn. This highlights the fact that ours is not the only use case for designing a CNL. Kuhn [55] notes that "the more one analyzes the variety of CNLs that exist, the harder it is to concretely define". Our discussion, like most of the other work quoted above, will be limited to CNLs designed for purposes of formalization and automation.

2.3 Linguistics

Perhaps ironically, much of the research being done in the field of NLP is not directly concerned with linguistics – the study of language. This is particularly the case in ML-based applications where models are simply fed large quantities of textual data and given labeled examples to drive the learning mechanism. In these cases, building a grammatically correct and semantically sound output is largely probabilistic. If a language model sees enough data, it will begin outputting text that conforms to the same linguistic rules as the training data. While this approach may be valuable for some use cases, processing legal documents into fine-grained formal representations requires deliberate incorporation of linguistic rules. Throughout this work, we will be carefully analyzing and constructing linguistic patterns, which consist of a large set of linguistic building blocks. In this section, we will provide some background, definitions, and examples of some of these building blocks. Much of the information presented in this section comes from Huddleston et al. [49].

2.3.1 Sentences, Clauses, Phrases

NL can be analyzed on a number of different levels. The two levels that interest us are syntactic analysis and semantic analysis. *Syntax* deals with how words can be assembled

into grammatically correct sentences. 'I saw a bird in the tree' is grammatical, but 'Saw I bird a in tree the' is not, and syntactic analysis can tell us *why* this is the case. *Semantics* deals with the *meaning* of a sentence. It tells us that at some point in the past, you experienced seeing a bird in a tree. A sentence like 'I saw the tree in a bird' is syntactically correct, but does not make semantic sense (excluding metaphorical considerations). We will be dealing with both types of analysis heavily in this work. A *sentence* is the largest textual unit that has syntactic structure. A sentence is made up of constituents, which can be grouped into categories and serve specific functions within a sentence.

The tree structure at the bottom of Fig. 2.5 shows the different constituents of the sample sentence. The sentence is split into two main constituents – the Noun Phrase (NP) and the Verb Phrase (VP). Constituents can also be made up of other constituents. The first NP functions as the *subject* of the sentence, and the largest VP functions as the *predicate*. Within that predicate, we have a Prepositional Phrase (PP), which contains another NP. This NP ('the seller') functions as the *prepositional object* of the PP. In general, we can say that a *phrase* is a sequence of words in which one word is a *head*. A VP has a verb as the head, a NP has a noun as the head, and a PP has a preposition as the head. A *clause* is traditionally considered as a grammatical piece of text that has a subject and a predicate. It can also be framed as the smallest piece of text that can *describe a situation*. When we speak of a clause, we will refer to a phrase that has a VP as its head.

2.3.2 Prepositions

We will not discuss the common word categories such as verb, noun, and adjective, but one category that requires elaboration is the *preposition*. A preposition is traditionally thought of as a word (or group of words) used before a NP to give extra information about something. For example, in 'the rabbit with the long ears', the preposition 'with' precedes the NP 'long ears' (the prepositional object), and gives extra information about the rabbit. The sentence 'I will go to the store in my truck at noon' has three PPs: 'to the store', 'in my truck', 'at noon'. All three prepositions (to, in, at) have a prepositional object (the store, truck, noon) and give extra information about my act of 'going'. In this use of the PPs, we say that the prepositions *license* a prepositional object, and that the prepositional object is the *complement* of the preposition.

Traditionally, prepositions have been characterized as *strictly* consisting of these words that license a prepositional object. If a word does not license an object, then traditionally it would *not* be considered a preposition. Consider the following examples that all use the word 'before':

- Let's eat before noon.
- I've never eaten squid before.
- Let's eat before it gets too late.

In the first sentence, 'before' licenses the object 'noon', so it would be consistently labelled as a preposition. In the second sentence, no object is licensed, so in this case 'before'

would traditionally be considered an *adverb*. In the third sentence, it introduces a new clause ('it gets too late') and would therefore traditionally be considered a *subordinating conjunction*. Some linguists argue that this traditional interpretation is awkward and that all three cases should consider 'before' to be a preposition, since the only difference is the *type* of complement that they license. In the first sentence, the complement is an object. In the second, there is no complement. In the third, it is a clause. We will adopt this broader conceptualization of a PP throughout this work, but we note that there are multiple interpretations, and different NLP tools that categorize text may use differing definitions. Fig. 2.7 shows how the three sentences are interpreted by the Natural Language Toolkit (NLTK) tree parser [2]. The first example is interpreted as a preposition (the tag IN refers to a preposition) in a PP. The second example treats 'before' as an adverb. The third labels it as a preposition, but there is no label for a PP on the subordinate clause.

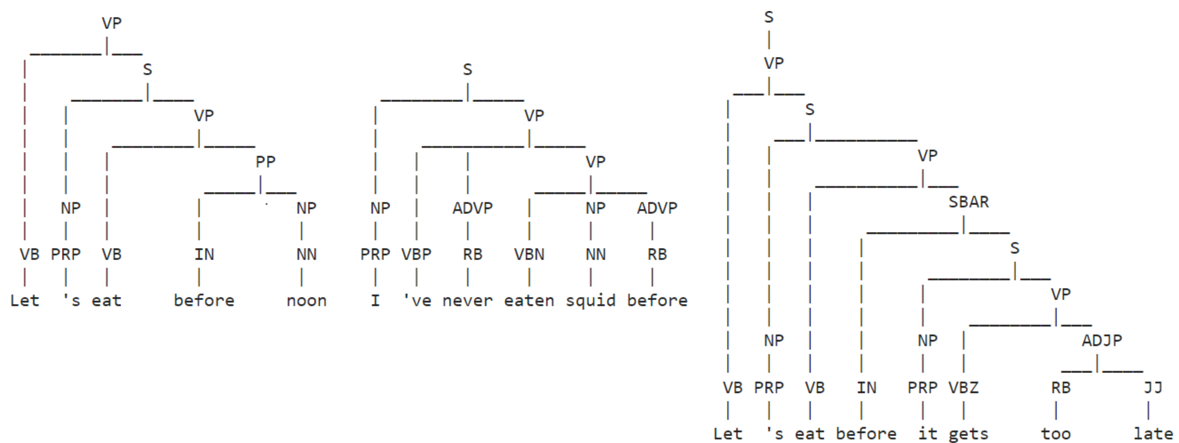


Figure 2.7: Three interpretations of 'before'

These differing grammatical interpretations will need to be addressed when we are looking for certain grammatical patterns inside NL text. For example, if we wanted to identify sentences such as 'I have never eaten squid before', we may need to search for cases where before is used as a preposition *as well as* cases where it is used as an adverb.

2.3.3 Adjuncts

Finally, we turn to the concept of the *adjunct*. Adjuncts are linguistic constituents that are *structurally dispensable*, i.e., they may be removed without affecting the grammaticality and meaning of the sentence. Adjuncts can be categorized based on the meaning they add to a sentence. Some categories of adjuncts, with examples (where the adjunct is in italics) include the following :

- **Manner:** She walked to the store *slowly*.
- **Locative:** The dog begged for food *under the table*.

[2] `https://www.nltk.org/howto/tree.html`

- **Temporal:** We should arrive at the airport *two hours before takeoff.*
- **Duration:** I read a book *for three hours* last night.
- **Frequency:** *Every Tuesday morning*, I go to the gym.
- **Purpose:** I took the longer route *to avoid traffic.*
- **Conditional:** I wouldn't go in there *if I were you.*
- **Instrumental:** She played us a song *with a fiddle.*

We can also classify adjuncts based on the various grammatical forms that they can take:

- **Adverb:** You must leave *immediately.*
- **Prepositional phrase:** You can have dessert *after you finish your vegetables.*
- **Noun Phrase:** They arrived *this morning.*
- **Finite clause:** I couldn't do it, *however hard I tried.*
- **Non-finite clause:** I kept my mouth shut, *to avoid giving any more offence.*

The function served by the adjunct in all of these examples is to give *more* information about the subject of the sentence. In other words, they *refine* the sentence of which they are a part. Furthermore, they can be omitted from the sentence, and it still retains a similar (though broader) meaning and is grammatically correct. While the concept of adjunct is well established, it is an open-ended classification with no clear boundary on how many types exist. Nor is it always clear whether a phrase plays the role of an adjunct. We used the example of 'under the table' as a locative adjunct, but consider the same phrase used in the following context: 'Please put the flowers *under the table*'. In this case, removing the 'under the table' yields the ungrammatical text 'Please put the flowers'. This shows that there are other complex linguistic factors at play, such as the type of verb, in determining if a given addition is optional.

2.4 Summary

Now that we've defined and discussed these important preliminary concepts, we can bring them all together to restate our problem in better-defined terms. Beginning with a NL contract template, we want to allow a contract author to make NL refinements to the template. Since the input is newly-generated text, we can impose restrictions on this input in the form of a CNL, which will allow for more reliable and unambiguous processing. The CNL must be expressiveness enough to represent relevant contract refinements, and also precise enough to remove ambiguities. The specific construction of the CNL will be addressed by our first three RQs. These CNL refinements will be processed into Symboleo operations, which will be applied to an initial formal specification `S(T)`,

resulting in a customized Symboleo specification S(C) that corresponds to the updated NL template. This processing may involve applying a combination of linguistic rules and reliable ML-based models to extract Symboleo concepts such as assets, domain events, and obligations.

In the next chapter, we will look at existing work related to these concepts to see how they are applied to problems that are related to our own.

Chapter 3

Related Work

Our specific area of research involves using a CNL to formalize legal contract refinements into a formal Symboleo specification, which can be used for smart contract monitoring applications. While there is a certain amount of existing research that closely aligns with this goal, we also want to address the general associated areas of CNLs and formalization of legal contracts. We will begin with a discussion on related formal representations of legal documents, so that we can understand where Symboleo fits in with the existing research (Section 3.1). We will then review relevant work related to CNLs, with emphasis on their application to the legal domain (Section 3.2). A particular challenge in this field is the representation of events, so we will discuss relevant approaches related to specifying and modelling events in a legal context (Section 3.3). Next, we will discuss research efforts towards contract formalization, focusing primarily on linguistic-based approaches, but we will also mention relevant ML-based ones (Section 3.4). Finally we will bring these general concepts of CNLs and formalization together and discuss research that involves using a CNL to partially automate the formalization of legal texts, which most closely aligns with our own work (Section 3.5).

3.1 Formal Representations

We survey the landscape of relevant ontologies in order to provide further clarification on the role an ontology plays with respect to this thesis. These ontologies often form the basis for formal specification languages including Symboleo. We will discuss some of these languages to better understand how Symboleo fits in to the existing formal specification and smart contract research fields.

3.1.1 Legal Ontologies

Semantic Web

A major driver in the development of ontologies in many domains, including the legal domain, is the Semantic Web [8]. The Semantic Web is based on the idea that the links

we use to navigate the World Wide Web should also include semantic meaning. Rather than simply linking between pages using hyperlinks, we can use more complex and precise structures to incorporate machine understanding into the web. This could facilitate the creation of applications that require interoperability between humans and machines, such as a legal information retrieval systems.

In order to encode understanding in machines, we need a uniform way of representing the data that we are working with. This need motivated the creation of the Web Ontology Language (OWL) [1], which provides standards for defining and relating ontological concepts. For example, if we had a concrete specification of a legal obligation, we could ask an information retrieval system to find all such obligations within a large set of documents, as well as their linked properties. Suppose that an obligation is defined as always having a *beneficiary*. In our semantic network, the obligations that are retrieved would always include this linked beneficiary, which may then link to further properties (e.g., perhaps every beneficiary has an address, which has a city, which has a population, and so on). OWL provides a framework for the construction of custom ontologies, which can then be shared across the web. Various legal ontologies have emerged in light of the Semantic Web.

DOLCE and CLO

Descriptive Ontology for Legal and Cognitive Engineering (DOLCE) is a foundational ontology that is part of the WonderWeb project [12], which aims to develop infrastructure for large-scale deployment of ontologies as a foundation for the Semantic Web. DOLCE was constructed with high-level concepts in mind, such as temporal and part-hood considerations. Consider a statue made of clay. Is it possible for both the clay entity and the statue entity to exist at the same time and in the same place? Or is the statue a *temporal phase* of the clay? This is an example of the high-level questions addressed by foundational ontologies like DOLCE and UFO [41]. Core Legal Ontology (CLO) [35] is one of many ontologies built on top of DOLCE. CLO identifies norm comparison as a key legal task, which could be important for measuring the impact of changing legislation. It defines concepts such as legal persons, legal roles, norms, and legal events, and creates an important distinction between *descriptions* and *situations*. This is relevant to the legal domain, where a *description* describes something normative (law or regulation), and a *situation* refers to facts about a given case. This approach of extending a foundational ontology like DOLCE into a more specific ontology like CLO is reminiscent of the relationship between UFO-L [39] and UFO [41], where UFO-L extends the notion of a *relationship* to include more specific *legal relationships*.

Legal Ontology Surveys

Surveys of legal ontologies have been performed by Van Engers et al. [91] and de Oliveira Rodrigues et al. [21]. These studies highlight trends in the development of legal ontologies and suggest fundamental and unique characteristics of the legal domain. A key insight is

[1] `https://www.w3.org/OWL/`

that norms in law are often under-specified in terms of world knowledge (i.e., common sense). They provide only a partial description of the world, and the rest is assumed to be known. As previously mentioned, the assumption of common sense poses a major problem for automation, and this provides strong motivation for the adoption of foundational ontologies. If the concepts in our legal domain are ontologically related to concepts in adjacent domains, then integrating into an ontological system allows us to fill in conceptual gaps created by the under-specified NL legal text. For example, consider a contract that specifies that a buyer must pay a seller. The concept of payment may seem commonsense to a human, but for a computer, the meaning must be formally encoded. Rather than equipping a legal ontology with the concept of a payment designed from scratch, we can link to a financial ontology, which specifies all that a computer needs to know about the concept of a payment (methods of payment, currencies, etc.).

The ontologies examined in these surveys vary in terms of purpose, granularity, and theoretical basis. Some have a very narrow purpose, which may be valuable for a specific domain, but have limited scalability outside that domain. Many of the ontologies share similar foundations to that of Symboleo. There is also agreement on sources of difficulty in formalizing legal texts. Some of these issues include the sheer volume of legal information that exists, the fact that normative texts are often subject to change, and the fact that legal terms can have different meanings in different domains and jurisdictions. This last fact may help explain why there are many different ontologies, rather than just a single shared one. While it may be beneficial to have a common specification for a 'legal obligation' across all legal documents on the web, the precise meaning of an obligation is very context-dependent, making these universal specifications challenging. For example, an obligation in the context of contract monitoring may have a different meaning than an obligation in the context of case law.

Legal Specification Languages

Legal Knowledge Interchange Format (LKIF) [46] is another legal core ontology, but it is not built on top of any specific foundational ontology. The goal of LKIF is to enable interoperability across various legal knowledge systems, aligning with the goals of the Semantic Web. Moreover, the specification of LKIF is built partly using a version of OWL. In addition to representing the legal domain, it also aims to integrate legal reasoning and inference. As expected with a core ontology, LKIF seeks broad coverage of the legal domain, and was constructed by surveying legal terms used across many domains.

LegalRuleML [3], like LKIF, explicitly follows the Semantic Web goal of interoperability across various web resources. It cites LKIF as an influence, but notes that is lacks the ability to handle more complex temporal aspects. As with other legal ontologies, LegalRuleML specifies concepts such as legal agents, events, and norms. The syntax used to model norms is influenced by various types of formal logic (first-order, deontic, defeasible), and a case study from Australia's National Consumer Credit Protection Act is presented as an example in the work.

3.1.2 Smart Contract Specification Languages

While Symboleo was largely written for the purpose of being applied to the development of smart contracts, it is important to emphasize that the goal of formalizing NL contracts precedes the realization of smart contracts. This relatively new application, however, provides a new source of motivation towards formalization, since formalization is a prerequisite for specifying smart contracts. A handful of relevant specification languages have been developed specifically for smart contracts.

Ladleif and Weske [56] attempt to unify these various languages in terms of the legal concepts represented (e.g., roles and obligations) as well as higher-level modelling concepts (e.g., actions, conditions, and instantiation). They found that most languages surveyed support the concepts of actions and general conditions, but many did not support the specific concept of a *temporal* condition.

Dwivedi et al. [24] present an ontology and specification language with the purpose of building smart contracts for collaborative decentralized autonomous organizations. The ontology is based on OWL and contains many overlapping concepts with Symboleo's legal ontology. They also address the issue faced by related efforts of allowing certain norms to trigger the creation of other norms (similar to the potential role of the trigger in Symboleo). A markup language to specify contracts (SCLML) is based on this ontology, and can be converted to Solidity [2] code for a smart contract implementation.

EFlint [89] is a domain-specific language for monitoring and controlling prescriptions found in legal documents. It has a similar foundation to Symboleo, supports the concept of a legal power to modify other legal situations, and emphasizes a contract flow based on legal actions between legal agents. The work also presents an example of formalizing norms found in the General Data Protection Regulation (GDPR) and further work applies EFlint to the healthcare domain [90].

SmaCoNat [75] is a specification language made specifically for smart contracts. The language is rooted in an ontology that provides terminology for agents, commitments, transactions, and resources. The goal is to make a language that is human readable, legally binding, and also executable. Therefore rather than having *two* separate specifications of a normative document (i.e., one in NL, the other in a formal specification language), SmaCoNat seeks to combine these into one. In theory, legal experts would be able to understand and write in the specification language. A key idea from SmaCoNat includes a restriction on the ability for programmers to give completely custom names to variables and operations, as opposed to enforcing more predictable and descriptive domain-related identifiers

Stipula [20] is a domain-specific language for legal contracts with a view towards concrete implementation in either a centralized or distributed system. The goal, like SmaCoNat, is to create a language that is sufficiently high level that it can be directly intelligible to legal professionals, and to this end it supports concepts such as type inference. It uses familiar legal concepts of obligations, prohibitions, and permissions, which include time-bound provisions and event triggers to create obligations. A simple bike rental contract is provided as an example in the paper.

[2] `https://soliditylang.org/`

3.1.3 Relationship with Symboleo

We've now seen a variety of legal ontologies, representations of legal concepts in general, and representations specifically for smart contracts. Many of these representations share links with Symboleo via a common ontology and/or legal foundation. Unification of the various ontologies is difficult since the specific meaning of terms is highly dependent on context. However, these commonalities provide support for potentially stronger links in the future or application of this work to other languages. The development of Symboleo arose out of a collection of requirements, which were not fully represented in any one specification language at the time it was developed [80]. These requirements include support for temporal concepts (time points, events, intervals), legal concepts (obligations, powers, roles), support for compliance monitoring, and support for automated reasoning and analysis. The specific concept of a legal power, which enables the modification of the state (e.g., suspension and resumption) of *other* obligations on the contract is notably missing from many existing Contract Specification Languages (CSLs). These are some of the key motivating factors for the creation of Symboleo as well as the justification factors for using it in the present work. Symboleo is also under active development, and recent developments in the Symboleo ecosystem include formal verification [68, 69] and code generation [73, 74] tools.

3.2 Controlled Natural Languages

The ambiguous interpretation of a NL statement results in difficulty in formalizing such a statement. A CNL, which can be consciously designed to restrict this ambiguity, can be seen as a stepping stone between unrestricted NL and a tightly restricted formal representation. We will now discuss important related work on CNLs, particularly CNLs designed for a legal context.

3.2.1 CNL Design Considerations

Upon analyzing a wide variety of CNLs, Kuhn [55] defines a classification system to help organize the CNLs and find patterns in their construction. Some classification metrics relate to the goal that the CNL is trying to solve (e.g., easier communication, facilitating formalization), and the domain that it serves (e.g., legal domain, industrial applications). The *Precision, Expressiveness, Naturalness, and Simplicity (PENS)* classification schema is also introduced, which evaluates a language based on four criteria. A precise language is one in which there is no room for ambiguous interpretation. An expressive language will allow one to express anything that could be expressed in an NL. Naturalness refers to how readable and understandable the language feels compared to the NL on which it is based. Simplicity is based on how many pages (a proxy metric for the number of rules) are needed to fully specify the language. A higher simplicity score means fewer pages.

For each CNL studied in the survey, a score from 1 to 5 was assigned to each of these four metrics. For example, English is scored with low precision (1), high expressiveness (5), high naturalness (5), and low simplicity (1). This score can be codified as P1-E5-N5-S1.

Basic English, an early CNL, is considered only slightly more precise and receives a score of P2-E5-N5-S1. A formal language such as propositional logic receives a completely opposite score of P5-E1-N1-S5; it is very precise, not very expressive (for example it does not allow for quantification), unnatural (compared with English), and it is simple to learn. We might expect a similar score for a formal specification language like SYMBOLEO.

The author notes several relevant correlations between these factors. Precision and simplicity are highly correlated; Languages with precise meanings tend to be simpler. Since we are aiming for a fairly precise CNL, this correlation may work in our favour. Unfortunately, there is also a negative correlation between naturalness and simplicity. If a language is made to look and feel like English, then it will tend to have more rules to specify, resulting in lower simplicity. As one might expect, there is also a negative correlation between simplicity and expressiveness, since more capabilities of expression will require more rules to specify the boundaries of those capabilities.

While Kuhn's survey focuses on CNLs from a very broad perspective, Wyner et al. [95] provide another source of important CNL properties and design considerations, with emphasis on CNLs that align specifically with our use case of formalization. A CNL should be easy to learn, unambiguous, and translatable to logic. We can see how these align quite nicely to some of the PENS criteria defined by Kuhn. Other important considerations mentioned include the following:

- **Extensibility of the language:** If a CNL is incorporated into a software, an important software engineering principle to recognize is that software must often change to meet changing demands. A CNL should therefore be designed to be extensible. We must keep this idea of extensibility in mind when we write the software that will process a CNL as well.
- **Mapping to a conceptual graph:** Mapping a CNL to a graphical representation would make it very amenable to formalization and automation. However, this will inevitably come into conflict with any simplicity requirements we have, as a complex language will likely result in the combinatorial explosion of a graph.

This discussion equips us with a concrete set of considerations and questions to ask in designing our CNL. An ideal language may score a 5 on all four metrics in the PENS schema. However, we expect to be constrained by the observed negative correlations. We want a sufficiently precise language so that it can be formally represented without any ambiguities, and we want the contract author to be able to clearly express useful contract modifications in a language that feels natural. These competing requirements mean we may need to accept the trade-offs inherent in CNLs. We can establish some general thresholds on some the factors. If the language feels unnatural or is too complex, then there is no pragmatic use case. To give a rough range of scores along the PENS classification, we aim for high precision (4-5), medium expressiveness (2-3), medium naturalness (2-3) and medium simplicity (2-3). With these goals in mind, it is valuable to consider other CNLs that have a similar score, a very popular one being Attempto Controlled English, which is assigned a score of P4-E3-N4-S3.

3.2.2 Attempto Controlled English

Attempto [30–32] was constructed for purposes similar to our own. It allows domain specialists to express specifications in familiar NL and combines this with the rigour of formal specification languages. The authors identify as being stuck between the over-flexibility of NL and potential incomprehensibility of formal languages. They note, however, that NL has the *potential* for being precise, if it is appropriately restricted. They also build off insights of previous CNL efforts, where reports indicate that they can be fully learned within a few weeks. Some key examples of ways that Attempto restricts NL include:

- Enforcing that verbs must be present tense and use active voice (as opposed to past/future tenses and passive voice).
- Fixing a subset of NL words and phrases with pre-defined functions (e.g., determiners, conjunctions, prepositions). These are contrasted with *content* words, which represent a more flexible category and includes nouns, verbs, adjectives, and adverbs. This distinction aligns quite closely with the distinction between open class and closed class words.
- Defining a simple sentence (e.g., subject + verb + complement) and allowing more complex composite sentences to be recursively built from these simple structures through predefined operations.

It is important to note that these rules introduced for Attempto are largely based on pre-existing linguistic constructs. They are simply choosing a set of useful and relatively precise conventions as the limits of their CNL, and not allowing for exceptions (of which there may be many in natural English). The result is a language that appears to be natural, but is actually a formal language that is readily convertible to first-order logic as well as discourse representation structure [52], another approach to formal semantics. It is therefore considered both human and machine-readable. In this sense, the goal of this thesis is very similar. We can take into account the linguistic structures used in Attempto's restrictions as we construct a CNL that is oriented towards Symboleo and the specific domain of contract event monitoring.

3.2.3 CNLs for the Legal Domain

Now that we have an understanding of some considerations involved in CNL design, as well as a concrete example, we can look at relevant research on CNLs specifically proposed for the legal domain. Finnegan [26] proposes a contract CNL for the purposes of contract drafting. While the potential of building software that makes use of such a CNL is mentioned, the primary motivation is to design a language that makes contracts more readable. This emphasizes the multi-faceted motivation for designing CNLs, particularly for the legal domain, where, as the author points out, language is often archaic and difficult to understand. Some concrete principles that are suggested include:

- Elimination of duplicate and redundant words (e.g., use 'to' instead of 'for the purpose of'. This especially applies to overly complex legalese. Contracts tend to use archaic phrases such as 'herein', 'in witness whereof', 'subsequent to', etc., and many of these can be replaced by much more common synonyms (respectively, 'in this agreement', 'signed', and 'after').
- Eliminate ambiguous terms (e.g., replace 'best effort' with a more specific obligation).
- Don't turn verbs into nouns (e.g., use 'conclude' instead of 'arrive at the conclusion'). This is also known as *nominalization*.

While this thesis focuses on the English language, there is also relevant research done with CNLs based on other languages. Hoefler and Bünzli [45] designed Controlled Legal German (CLG), a CNL for representing legal norms found in Swiss statutes and regulations. The key motivation was to have a formal representation of legal information in a structured knowledge base. Key techniques involve restricting syntax and semantics to prevent ambiguity. The deontic concepts that it focuses on include obligations, permissions, and prohibitions. The result is a set of definitions about what constructions CLG sentences can consist of and what formal logical representations they map onto. A key missing element in the paper that is relevant to our work is the specific inclusion of events, though the authors note that this will be included in future work.

3.2.4 CNLs for Requirements Engineering

We can also look to the field of Requirements Engineering for relevant work, where CNLs have been similarly used to simplify formalization while maintaining readability. Pires et al. [72] integrate a customized version of Attempto into a tool-supported process that can capture requirements knowledge from NL descriptions. They also combine this with an ontologically-based process for capturing relevant domain knowledge. The specific incorporation of domain terminology into a CNL is a technique that we will also make use of.

Barros et al. [6] propose UcsCNL, a CNL for creating use case specifications. The CNL is applied to a specific format of test cases, which contains an initial condition, a user action, and a system response. Each of these three components has a distinct set of patterns, which are in turn built of components rooted in a concrete grammar. Parts of speech (Parts of Speech (POS)) such as nouns and verbs receive certain restrictions and some allow for dynamic user input. For example, nouns and verbs represent an *open* class, which means the list of words that function as nouns and verbs is quite dynamic. *Closed* word classes include prepositions and conjunctions, which have sets of words that are relatively fixed. Since this CNL is still fairly dynamic, the authors implement their own POS-tagger and parser to probabilistically formalize the CNL input.

Fockel and Holtmann [27, 28], Holtmann et al. [48] propose a methodology and a CNL based on a set of requirement pattern templates. For example, a common requirement template they find is: 'When the event <event> occurs within the system <system> [and the condition <condition> is fulfilled], then the function <function> is (activated |

deactivated)'. This template tightly controls the format of a requirement. This approach may be very useful when applied to a specific use case, but likely would not scale to other domains. Within the template itself, there are further terms (e.g., event, system) that must also be concretely defined.

3.2.5 Formal Languages

The PENS classification schema for CNL shows how languages such as NL and predicate logic can exist on the same spectrum. We could therefore interpret many of the representations from our previous section on this spectrum, where they would be much closer to the formal languages (e.g., higher precision, lower naturalness). Formal languages like Stipula [20] and SmaCoNat [75] are designed to be directly intelligible to legal professionals. Since the key consideration for a formal language will be precision, other negatively correlated factors may necessarily need to be sacrificed, especially naturalness. These competing needs can potentially create friction for users who are required to learn this new language.

We do not take this approach with Symboleo, as the terminology would be unduly complicated for legal professionals. Instead of a single language that aims for both precision *and* naturalness/expressiveness, we keep these competing considerations *separate*. Symboleo is created primarily for precision, with naturalness and expressiveness as secondary considerations. Our CNL will be created primarily for expressiveness and naturalness, but with a minimum threshold on precision, in that we want to reduce any ambiguous interpretations. This keeps each language more focused. The goal of Symboleo is to precisely represent legal contracts, and the goal of the CNL is allow contract authors to comfortably and naturally customize contracts. Our task will be to identify common relationships between NL and Symboleo in order to *construct* this CNL.

3.3 Event Specification

Symboleo is based on the idea that instances of obligations and powers go through state changes, which are triggered by events. Since events are central to the design of Symboleo, we will need a structured way of representing them in our CNL. This presents an immediate difficulty, since the specification of an event is not trivial. A dictionary definition of 'a thing that happens' [3] does not offer much more clarity, so in designing our CNL we will need to constrain this definition.

3.3.1 FrameNet

FrameNet [4] is based on the idea central to Frame Semantics that the meanings of most words can best be understood on the basis of a semantic frame, which is a description

[3] `https://www.merriam-webster.com/dictionary/event`

of a type of event, relation, or entity and the participants in it. We illustrate with an example. Consider the act of fishing. This is considered a common human experience, so it is captured as a *frame* in FrameNet. The notion of fishing only has meaning if certain roles are filled by concrete entities. In order for the act of fishing to make sense, we need to know things like *who* is doing the fishing, *what* is being fished, *where* the fishing is taking place, etc. The roles of the fisher, the fish, and the location are *frame elements* in the *fishing* frame. The FrameNet definition of fishing is: 'To seek water-born living flesh through the use of effective tools.' These three core frame elements are defined as follows:

- **Hunter:** The Hunter is the agent hoping to obtain Food;
- **Food:** Food is the sought-out item by the Hunter;
- **Ground:** This is the entity to which the Hunter pays attention. It is referred to as Ground because it serves as the background or context for the Food.

Frames can also be related to each other. In fact, the 'Fishing' frame is a more specific instance of the 'Hunting' frame, which contains the same frame elements. Since frames attempt to capture the endless and dynamic varieties of human experience, the system is inherently incomplete. However, if it handles events that are related to our domain of legal contracts, then we may be able to leverage it. Furthermore, it provides us with a more concrete way of thinking about how events might be specified. In our distinction between ML-based and linguistic-based NLP techniques, FrameNet is considered a linguistics-based approach. An interface to FrameNet is built in Python, which provides access to the various frames and their elements contained in FrameNet [4].

3.3.2 ACE

Another important endeavour in concretely specifying events that shares similarities to FrameNet is the Automatic Content Extraction (ACE – not to be confused with Attempto Controlled English – which is sometimes denoted with an identical acronym) from the Linguistic Data Consortium [58]. The ACE dataset consists of tagged events along with guidelines for how events are specified. While it primarily deals with how events might be extracted from unrestricted NL, this has clear relevance for how we might specify events in a CNL. According to ACE, an *event* is defined as a specific occurrence involving participants. A key concept involved in event identification are the event trigger, which is a word in the text that most clearly expresses the event. For example, in the sentence 'They have been married for 3 years', the trigger word is 'married'. This recalls the trigger phrase approach discussed in linguistics-based NLP techniques in Sec. 2.2.3. It is tempting to think that the trigger may always be a verb, but that is not the case. For example, in the sentence 'The explosion claimed at least 30 lives', 'explosion' (a noun) is identified as the trigger word, as it implies the verb/event of 'exploding'. The notion of a trigger is further complicated when we encounter more complex sentences with multiple verbs. Other properties that are considered in the specification of events include:

[4] `https://www.nltk.org/howto/framenet.html`

- **Polarity:** A ‘negative’ polarity event is one that specifically *doesn’t* happen;
- **Tense:** e.g., past, present, future;
- **Genericity:** Is the event a single specific occurrence (e.g., I watched a movie) or does it generally refer to a finite set of occurrences (e.g., I watch movies)?
- **Modality:** This property describes how definite the event is. The range includes assertions, commands, beliefs, and hypotheticals.

3.3.3 Frame-based Event Specification

Just as FrameNet uses the concept of Frame Elements to specify a frame, ACE uses an equivalent notion of an event argument. A relevant note is that ACE conceives of an event as involving *participants*, and the event arguments are often going to be these participants. This is relevant because the events that we are interested in monitoring for contracts are typically going to be occurrences involving one or both parties of the contract. The potential correspondence between FrameNet and ACE is well-documented, and efforts have been made towards event detection systems that unify them [86].

In addition to FrameNet and ACE, there are other initiatives for defining events specifically in the legal domain. Navas-Loro and Santos [64] compare and analyze these initiatives with a hypothesis that there may be ways of unifying them into a common framework for specifying legal events. This recalls our earlier discussion about goals of the semantic web, and the authors identify the familiar issue that even within the legal context, there are more fine-grained contexts where legal events can have different meanings. Examples of verbs that have very different contextual meanings include ‘appeal’, ‘argue’, and ‘submit’. While consensus across the entire legal domain remains unsolved, the framing of events as having two key components (the trigger word and the roles) aligns with Symboleo.

In Symboleo, events are specified in the domain model with a descriptive name and a list of domain properties. The name can correspond to the trigger, and the the domain properties correspond to the event arguments. For example, consider the ‘Delivered’ event from a sample contract: `Delivered isAn Event with item: Meat, deliveryAddress: String, delDueDate: Date;` The three domain properties include the item to be delivered, the delivery address, and the delivery due date. When this event is present in an obligation, we also add the roles of the debtor and creditor. These would respectively correspond to the agent that is *making* the delivery and the agent that is *receiving* the delivery. Now consider the core frame elements on the ‘Delivery’ frame in FrameNet:

- **Deliverer:** This is the person who completes the movement of the Theme after having transported it.
- **Goal:** The end of the path and intended goal of the sending.
- **Theme:** The objects being delivered.

- **Recipient:** This is the Recipient of the sent Theme.

There is a clear correspondence between the frame elements from FrameNet and the event arguments in the Symboleo specification (Deliverer with debtor, Recipient with creditor, Goal with deliveryAddress, Theme with item). The delivery due date is an extra addition in the Symboleo specification. This correspondence suggests that the FrameNet/ACE approach to event specification is valid for our purposes. The next important question is how we represent this in a CNL. We need to define rules for a CNL that allow us to capture the event trigger and its elements in a way that is precise but also natural. We will explore this in detail later on, but for now let us consider one way of representing our delivery obligation in NL: 'The *debtor* must deliver the *item* to the *creditor* at *deliveryAddress* before *delDueDate*.' The *debtor* element acts as the subject of this sentence, the *item* is the direct object, and the other elements are specified using prepositional phrases. This example suggests that we may need to incorporate these basic linguistic elements into our CNL in order to represent events with the desired level of expressiveness.

El Ghosh and Abdulrab [25] propose a model for specifying events in the legal domain, which is based on analyzing ontologies that are relevant to the legal domain, including UFO-L. The authors identify a set of requirements for modelling events in the legal domain, some of which include representing the following:

- Mereological (part-whole) relationships;
- Temporal relationships;
- Participation of objects in events;
- Changes promoted by events.

Many of these considerations align with provisions in Symboleo, which captures temporal relationships through the use of predicate functions, participation of objects in events through the use of declared roles and assets, and changes arising from events through the use of triggers and antecedents. Their work is a more *formal* model of how legal events are represented, and it thus validates the characterization of events as represented in Symboleo. It also suggests considerations for how legal events may be represented in NL. For example, the participants may be filled by roles or assets, which may take the form of noun phrases. Events can be linked temporally through the use of temporal keywords (before, after), and changes may be captured by the use of conditional keywords (if...then...).

3.4 Formalization

The key motivation for the construction of a CNL is to facilitate formalization. By making the NL contract refinements conform to a CNL, we remove ambiguity from the interpretation, thereby making formalization more predictable. The challenge is

to maintain desirable NL qualities such as expressiveness and naturalness so that it is amenable for the contract author. For this reason, we cannot make the CNL *overly* precise, and thus we will still need to apply more complex formalization techniques. We will now look at related work that attempts to formalize various types of normative NL documents into formal representations. Some of these representations have already been discussed. In many cases, the tasks we discuss did not have the luxury of using a CNL, but instead had to work with pre-existing and largely unrestricted NL. In general, this will make the formalization processes less reliable, but the techniques employed will still be relevant for our purposes.

3.4.1 Manual Efforts

We begin by looking at some notable efforts for *manually* formalizing legal documents, since the algorithmic nature of these approaches anticipates future automation. Breaux et al. [14] propose a process called *semantic parameterization* to derive semantic models from privacy policy documents. This approach begins with a set of NL privacy goals that are manually parsed into a restricted CNL. This involves simplifying complex normative statements into more atomic activities, consisting of a single action and a single actor. Once the goals are expressed in this simplified form, semantic roles required to construct a norm can be more easily extracted and then formalized. The formalized policies can be loaded into a computer to be systematically queried, compared, and analyzed. Breaux and Antón [13] further expands on this work by refining the models to account for other types of linguistic complexity (e.g., disjunctions, negations, and conjunctions). We note the presence of several implied sub-tasks in this workflow, such as:

- The intermediary conversion from pure NL to a more restrictive CNL;
- Fitting different terms from these restricted statements into templates that constitute a complete norm;
- Further refinement on these norms to account for additional linguistic complexity.

Libal and Steen [57] describe a methodology for validating legal knowledge bases inspired by Behaviour-driven design. They introduce a suite of tools to support this methodology, which allows non-technical experts to interface with formal legal knowledge. The formalization process itself is done manually by technical and legal professionals with the help of a graphic user interface. The formalized knowledge can then make use of the benefits that come with formalization, such as automated reasoning and model-checking. A manual approach to formalization is certainly not as scalable as the automated approaches that we will see, but they may allow for a gradual transition to further automation. For example, the interface may make use of NLP techniques to highlight suggestions for attributes of target concepts. The user interaction with this system could be analyzed to help gradually improve the performance of the suggestion mechanism, perhaps eventually to the point of full automation.

We now turn to automated formalization approaches, which can be roughly divided along the two tendencies we've already discussed: Linguistics-based and supervised ML-based. Linguistic-based approaches involve incorporating linguistic knowledge into the system, whereas ML-based approaches learn a model by training on a labelled dataset.

3.4.2 Linguistic-based Approaches

Using Frame Semantics

Frame-based techniques used in formalization are influenced by frame semantics and the FrameNet project. The Norm-in-Rete (NIR) project [29] was initiated by the Italian Government with the goal of more organization, transparency, and coherency in the Italian Legal System. It involved standardizing many aspects of the legal system, including the creation of an XML-like description for normative documents. The work by Biagioli et al. [9] involves identifying legal provision types and their arguments in legal documents within the NIR framework. The target representation is a set of custom frames for each provision type. There are 11 provision types, including prohibition, obligation, permission, and penalty. The arguments vary for each provision type. As an example, the *obligation* provision requires an addressee, an action, and a third-party. Consider the following NL sentence and its formalization: *The Member State shall pay the advance within 30 calendar days of submission of the application for advance payment*

Frame `obligation`

Addressee `The Member State`

Action `Pay the Advance within 30 calendar days of submission of the application for advance payment.`

Third Party `NONE`

The approach begins by passing the input text through a standard NLP pipeline, which is followed by a semantic annotation step. Each provision type is a frame whose elements, or slots (e.g., addressee, action, or third party), are filled with textual material that matches their conceptual roles. A supervised ML classifier is used to classify the paragraphs into their provision types. This highlights an example of how linguistic-based techniques and ML-based techniques can be combined. The subsequent extraction of arguments for filling the slots is based on linguistic patterns. For example, the addressee of an obligation is typically the syntactic subject of a sentence. The main action is often denoted by an infinitival clause. Properties like punctuation and dependency relations are also incorporated into the automated classification. It's important to note that these patterns are specifically for Italian legal texts, and other languages may have very different linguistic conventions for expressing normative texts.

Navas-Loro et al. [65] introduce ContractFrames, a frame-based approach for formalizing contract event flows into a specialized reasoning system called PROLEG. Other

approaches discussed in this section focus on *normative* concepts. This one, however, focuses on events that alter the *status* of a contract, which is also relevant for SYMBOLEO. Three specific contract events are established – buying, selling, and rescinding. Each event has a set of associated slots, for example 'date of purchase' 'currency', and 'price'. It uses a standard NLP pre-processing pipeline, looks for target trigger words for each of these events, and performs pattern matching to fill the slots. As an example, consider the following sentence and its formalization: *PartA sold landL to PartB by contractC for 20,000 dollars on October 13, 2017.*

```
agreement_of_purchase_contract(partB, partA, landL, 20000, 2017 year 10
month 13 day, contractC)
```

In these frame-based approaches, it is clear that certain slots may be much easier to fill than others. Information corresponding to dates or numbers is relatively simple to extract from the processed input text compared to more complex concepts like actions or conditions. Complexity would also increase if the number of target concepts were to increase. Separate rules would need to be identified for each type as well as for all slots associated with those new types.

Formalizing Requirements

We have already noted the close connection between the legal contract domain and the requirements engineering domain, and once again, we can look to this field for relevant work related to formalization of normative text. The benefits of formalizing NL requirements for software are familiar – ambiguity is removed from the requirements, and it facilitates rigorous testing, helping to ensure expectations are met.

Wyner and Peters [94] focus on extracting rules from a set of blood bank testing requirements into a RuleML representation (a precursor to LegalRuleML). Properties of the formal rules include an agent, main verbs, exception clauses, deontic modals, and conditional sentences. Consider the following sentence and its formalization: *Required testing must be performed by a laboratory registered in accordance with part 607 of this chapter and either certified to perform such testing on human specimens under the Clinical Laboratory Improvement Amendments of 1988.*

Agent 1: Required testing

Main Verb: Performed

Deontic Modal: Must

Agent 2: A laboratory

Conditional: registered in accordance with part 607 of this chapter and either certified to perform such testing on human specimens under the Clinical Laboratory Improvement Amendments of 1988

Pre-processing is done using the Stanford NLP parser [5]. Templates to identify the rules are generated using a software called GATE [6], which enables linguists and text engineers to build linguistic patterns from a corpus of text. Trigger words are identified to suggest the presence of certain properties. For example, an exception is triggered by the terms 'except', 'unless', 'other than', etc. A linguistic pattern is applied to confirm the presence of the concept. The pattern for an exception might consist of an exception trigger followed by a prepositional phrase or a noun phrase. These phrase types were extracted in the pre-processing step by the Stanford Parser.

Zaki-Ismail et al. [96] also address the general requirements space by introducing the RCM (Requirement Capturing Model) technique. This process formalizes NL requirements into the RCM format, which has notable parallels with the SYMBOLEO specification, such as temporal aspects. Target concepts include triggers, conditions, and time intervals. The formalization involves standard pre-processing steps, with the notable addition of *closed word unification*. This is where English words that hold the same function are unified and converted to a single representative word. For example, the words 'whenever' and 'once' can be replaced by 'when', echoing techniques we saw in the design of CNLs. Reliably restricting the NL in this way can make processing easier at later stages. The extraction of components begins with identifying the main verbs using the dependency parse, which is corroborated with WordNet [7].

3.4.3 Other Linguistics-based Approaches

Sleimi et al. [83] seek to reconcile the various legal formalizations into a harmonized ontological framework and introduces a process to convert NL traffic laws to this representation. 18 target concepts are identified, including actor, condition, and modality. The study focuses on extracting these attributes from NL text using a familiar pattern-based approach, but in this case the patterns are partly derived from an expert-annotated dataset. The input text underwent standard NLP pre-processing to produce constituency parses, which were then matched with the generated patterns. Trigger phrases are also incorporated, for example with words such as 'driver, 'officer', and 'inspector' suggesting the presence of an actor. A rule for identifying an actor is the presence of such a trigger word acting as a noun phrase as well as being a subject dependency.

Ash et al. [2] focus on extracting norms from collective bargaining agreements to help summarize gains and losses for the various parties when the contract changes. They follow the standard formula of pre-processing, trigger phrase identification, and pattern-matching. The target concepts include obligation, prohibition, permission, and entitlement. Since the work operates in the domain of labour contracts, the contracting parties are restricted to four subjects: worker, employer, union, manager. In general, being able to reliably restrict the expected language of the input makes it easier to extract the target concepts. Once the target concepts and their beneficiaries are extracted, a scoring mechanism is applied to summarize the authority of each party. Obligations and prohibitions will reduce

[5] `https://nlp.stanford.edu/software/lex-parser.shtml`
[6] `https://gate.ac.uk/teamware/`
[7] `https://wordnet.princeton.edu/`

an agent's authority, while permissions and entitlements will increase it. The work was applied to study the impact of the 2005 auto industry crisis.

3.4.4 ML-based Approaches

While some of the linguistics-based approaches we've discussed use elements of ML, they primarily rely on identifying concepts using linguistic analysis. We now look at a handful of approaches that primarily use supervised ML for extracting and formalizing legal concepts.

O'Neill et al. [66] employ both traditional ML algorithms as well as more modern neural networks to the extraction of norms in the financial legal domain. Their dataset consists of about 1300 expert-annotated sentences from public financial contracts. Experiments with traditional approaches (logistic regression, SVM, decision trees) show satisfactory performance, but this is greatly improved by incorporating deep learning techniques such as LSTM, which is able to build a sense of context based on the ordering of the words in the text. The main modalities of interest include obligation, prohibition, and permission. Chalkidis et al. [16] builds on O'Neill's work by exploring updated deep learning techniques and using a larger dataset of English service agreements.

Returning again to the alignment between the legal domain and the requirements engineering domain, Sainani et al. [76] focus on extracting obligations from software engineering contracts. The dataset consists of 20 such contracts, resulting in nearly 18K sentences. A variety of ML techniques are tested, with the notable inclusion of BERT-based models. The authors experiment with pre-training BERT on a large legal corpus so that contextual knowledge encoded by the model includes legal language. BERT-based approaches that use custom pre-training on legal texts have also been explored in other research [17].

3.4.5 Bridging Linguistics and ML Approaches

While these ML-based approaches achieve strong results, it is important to note that, in most cases, the goal is to simply identify a *span* of text that corresponds to a normative concept. This can be contrasted with some linguistic-based approaches, where the goal is to construct a more fine-grained representation of a norm, including the specific action, the parties involved, etc. This distinction is very important, as some use-cases (including contract monitoring) require the more formal representation. Simply annotating the spans may be valuable in legal contract review applications, where a legal expert is reviewing long documents and wants to quickly find potential mentions of obligations. The work of Funaki et al. [33] suggests that it may be possible to bridge this divide. Using an expert-annotated dataset of contracts, the researchers build a model that extracts not only rights and obligations, but also more salient information such as the contracting parties as well as conditions. Essentially, it uses a simple linguistic frame as a target representation, and has a ML classifier predict which span of text best fits within each slot.

The main obstacle towards any incorporation of a ML-based model is the need for a labeled dataset. While this may be a future consideration for Symboleo, it is possible

(and common) to leverage existing labeled datasets to new domains, and we re-emphasize that ML-based approaches actually underlie many of the pre-processing steps in the linguistic-based approaches we discussed. This approach is the basis for the linguistic classification of two very popular NLP libraries in Python, NLTK [8] and SpaCy [9]. Both libraries contain a wide variety of tools that handle this linguistic functionality and more. NLTK is focused on providing a wider range of linguistic resources that can be combined and configured for a diverse set of tasks, making it an important research tool. SpaCy is more focused on delivering strong results for a set of linguistic tasks and has preferences for its internal configurations, making it more accessible for business cases. It is, however, still very customizable.

3.4.6 Application to Symboleo

Since our problem involves newly-generated input into the system, we can take advantage of using a CNL to make the formalization process much more (but not *completely*) predictable. In general, the more restrictions we add into the CNL, the easier the formalization will be, but the less expressive it will be. Less expressiveness may result in less usability. Since expressiveness and naturalness are important considerations for our CNL, many of the techniques highlighted in this section may be relevant for our work. In particular, ML-based approaches for identifying certain entities or for semantic labelling could be incorporated into the expressive components of our CNL. These ML-based techniques may also become relevant as labeled datasets are developed for the Symboleo ecosystem.

Another important takeaway from this section is the relationship between *our* goal of *constructing* NL patterns for a CNL, and the goal in these approaches of *extracting* patterns. These goals are closely related, and the techniques used for *extracting* NL patterns may be applied to the construction process. For example, many approaches described above make use of trigger words and grammars to identify legal concepts such as obligations. If we were building a CNL to express these same legal concepts, we can use these same trigger words and grammars as *part of the CNL*. We also recall that this work is situated with a view towards *full formalization* of Symboleo contracts, and many of these approaches may become even *more* relevant in related future work.

3.5 Contract Customization

Now that we've addressed the research areas of formal representations, CNLs, and formalization approaches separately, we can look at specific cases where some of these concepts come together, which results in work that is most closely aligned with our own. The following related work focuses more specifically on the formalization of legal contracts into Symboleo (or similar languages) using linguistic-based techniques and/or CNLs.

[8] `https://www.nltk.org/`

[9] `https://spacy.io/`

3.5.1 GaiusT and ContracT

Manual formalization methods that lay out precise algorithms such as semantic parameterization [14] anticipate partial automation. As software companies attempt to comply with more and more regulations, the process of manually performing this formalization becomes tedious and error-prone, which motivates the goal of developing tools to partially automate the process.

Kiyavitskaya et al. [53] and Zeni et al. [97] developed GaiusT to this end. GaiusT is a web-based tool that focuses on extracting legal concepts such as norms from legal documents. A norm is characterized by a *norm-kernel*, which consists of properties such as the legal modality (obligation or right), conditions, and exceptions. GaiusT extends the existing Cerno semantic annotation functionality [54] to specifically handle legal documents and legal concepts. The core functionality involves generating a parse tree from a NL text and applying syntactic and semantic rules to infer annotations of the target concepts. This includes the familiar techniques of searching for trigger phrases and rule-based patterns. GaiusT is a full-stack tool, including database storage, a component for evaluating the performance, as well as a graphical user interface. This tool uses a *partially* automated process that requires input from the user, so the metric of interest is whether using this tool increases efficiency over a *completely manual* process. Experiments with GaiusT on HIPAA and an Italian Accessibility law show positive results.

ContracT [84] is heavily influenced by GaiusT and specifically targets legal contracts. This tool adopts the SYMBOLEO specification language and focuses on the sub-task of annotating powers and obligations. The paper outlines a four-step process for complete formalization of NL norms into SYMBOLEO:

1. Structural and semantic annotation to identify legal concepts in a text;
2. Identification of relationships among concepts;
3. Contract domain modelling;
4. Generation of the formal specification.

The first step of identifying the legal concepts is based on the familiar principle of identifying trigger terms and corresponding linguistic patterns, and can be expressed in an EBNF grammar. A simplified subset of this grammar is presented for illustration in Listing 3.1.

Listing 3.1: Grammar used in ContracT

```
<Obligation> := (<Party> | <Role>) & [<Party>|<Role>] & (<
   NecessityM> <Action> <Asset>) & <Situation>{1,3}

<Party> := ("Mr" | "Mrs") <Text> | <Text> ("Inc" | "Ltd")

<Role> := "supplier" | "buyer" | "landlord" | ...

<NecessityM> := "have to" | "has to" | "must" | "shall" |...
```

This suggests that a party can refer to a person (e.g., Mrs Smith) or an organization (e.g., Meat Suppliers Inc.). A role must be from a list of pre-defined string literals. The concept of necessity is indicated by trigger words that suggest the presence of an obligation. This grammar can be customized depending on the domain. For example, if our domain is apartment rentals, we may limit the possible roles to 'landlord', 'tenant', 'property manager', etc. The result of this pattern-based approach is a version of the input text that is annotated with terms defined in the grammar. This could then be fed into a system that handles step 2 of the formalization process. In this sense, the steps defined here are sequential, in that the output of a given step is the input for the subsequent one. All of these steps may begin as manual processes and have the potential to gradually be automated as technologies improve or the software is adapted to specific domains. A fully automated formalization process may then be a concatenation of all of these sub-tasks.

While not explicitly described as a CNL, the patterns that make up the grammar give us a useful starting point for designing a CNL that corresponds to SYMBOLEO. If we want our CNL to be able to express a certain legal concept, then we can model it off of these patterns. While our goal is not for the CNL to express full obligations, but rather *refinements* of obligations, we can use these patterns for other legal concepts that we need to represent, and it also provides a blueprint for future work that *may* incorporate the specification of full obligations in a CNL.

3.5.2 Contract CNL to Deontic Logic

Pace and Rosner [67] make a case for translating a CNL for contracts into a temporal deontic logic. The authors correctly identify the problems of ambiguity, complexity, and context sensitivity associated with using an unrestricted NL, which motivates the creation of a CNL. The authors specify formalisms for various legal situations using deontic logic. Consider the NL obligation: *Upon accepting a job, the system guarantees that the results will be available within an hour unless cancelled in the meantime.*

This sentence highlights two examples of ambiguity. First of all, the 'within an hour' may refer to the results or the guarantee. This is an example of *attachment ambiguity*, and often occurs on prepositional phrases such as this. Secondly, the cancellation may refer to the results or the job. The CNL devised in the paper addresses these issues by the introduction of identifiers and enforcement of a simplified event structure. In addition to the obligation of guaranteeing results, there are three separate events specified in this sentence (accepting a job, results becoming available, and cancellation). An event structure of agent-action-object is enforced, which also implies that the agent must be specified in each action. This is done by introducing identifiers such as SYSTEM, SOMEONE, and JOB. The three events could be formatted as follows:

- SYSTEM accepts JOB;
- SYSTEM make available results of JOB;
- SOMEONE cancels JOB.

Constructing further patterns around these simplified event structures can result in the full unambiguous CNL specification as follows: *if SYSTEM accepts JOB, then during one hour it is obligatory that SYSTEM make available results of JOB unless SOMEONE cancels JOB.*

Recalling the metrics from the PENS scale for CNLs, we note that while this is indeed more *precise* than NL, it is considerably less *natural*, highlighting a tradeoff between potential CNL desiderata. Since the language is now sufficiently precise, it can be represented using deontic logic. The types of NL statements addressed by this work are similar to those that we are working with, particularly since our focus will be on the *events* that make up the norms of a contract.

3.5.3 Custom Contract Templates with a CNL

Tateishi et al. [88] propose a technique to automatically generate a smart contract from a CNL contract using a template-based approach. For a given NL contract domain, a contract document template is manually created, which contains parameters that can be customized using a CNL. Given this template and the CNL arguments, a formal model is automatically generated. From there, it is translated into executable smart contract code. The contract template is created with a set of parameters, each of which must adhere to a specific grammar. For example, a sales contract template may contain the following text: '`The seller will ship the item [SHIPPING_LEAD_TIME] from the date of order.`' The parameter here is SHIPPING_LEAD_TIME, and a grammar specifies what is allowed to be input into this variable, a sample of which is shown in Listing 3.2.

Listing 3.2: Grammar used in a template-based approach to contract customization (Tateishi et al. [88])

```
<SHIPPING_LEAD_TIME> := <PERIOD>

<PERIOD> := "within" <WITHIN_DAYS> ("day" | "days") "if" <
   FROM_EVENT_DATE>

<WITHIN_DAYS> := NUMBER

<FROM_EVENT_DATE> := <EVENT_ARRIVE> | <EVENT_ORDER>

<EVENT_ARRIVE> := "arrival" | "arrival of the item"

<EVENT_ORDER> := "order" | "order of the item"
```

Therefore the phrase *within 12 days of arrival* is valid for this grammar, but *within 12 days of arriving* is invalid. If the input for a parameter does not adhere to the grammar, then it will fail, and the user may be prompted to enter a different input. In this way, we get a CNL that restricts the input to the system. The CNL text is then formalized into the Domain-Specific Language for Smart Contracts (Domain-Specific Language for Smart Contracts (DSL4SC)). Like SYMBOLEO, this specification language can represent contract

states and sequences of events that alter these states. A DSL4SC contract specification contains a set of protocols, properties, and rules, which specify what happens when certain events take place. For example, the formalization below shows that when a shipping event occurs, the shipping date is updated accordingly, if the present date is before the deadline:

```
on ship
    when { _event.data.date <= SHIPPING_DEADLINE }
    do { _data.shippingDate = _event.data.date }
```

Note that the SHIPPING_DEADLINE parameter is a parameter for the formal model, and is *not* the same as the SHIPPING_LEAD_TIME parameter on the contract template. The goal is to *map* the contract template parameters (SHIPPING_LEAD_TIME) to the formal model parameters (SHIPPING_DEADLINE). In this case, we would simply need to extract the number of days specified by the contract author in the shipping SHIPPING_LEAD_TIME parameter, and add that to the date the contract takes effect, giving us the SHIPPING_DEADLINE date value. In this way, the formalization is framed as a mapping problem: An argument in the contract template is mapped to an argument in the formal model. The contract author should only ever interface with the contract template language, and the technical details of the formal specification remain hidden. The template-driven approach with a CNL approach is very similar to the framing of our own problem. A key aspect of the approach described in this paper, however, is that for any parameter that is introduced (e.g., SHIPPING_LEAD_TIME), a *unique* and suitable grammar must be carefully specified. In our work, we aim to create a more general CNL that could be applied to *all or most* of the contract parameters.

3.5.4 Relevant Work from Requirements Engineering

Buzhinsky [15] performs a survey of approaches to requirements formalization from NL. He broadly categorizes the approaches into direct approaches (i.e., linguistics or rule-based) and statistical (machine learning) approaches, a distinction we are now familiar with. One of the direct approaches mentioned is a *pattern-based* approach, where generalized descriptions of commonly occurring requirements are grouped into patterns (a CNL) and predictably mapped to a formal specification. The author observes that this is only a viable solution if the requirements are prepared according to specific guidelines. This is indeed the case with our work, and this description is very much aligned with our proposal. The important question here is *how these patterns are determined in the first place.*

To help answer this, we can take cues from *other* formalization strategies mentioned by the author. The mechanisms used to detect patterns in unrestricted NL can guide us on how to *construct* these patterns. In some statistics-based approaches, the task is treated like a typical language translation task where a model learns the translation from a set of labeled examples. In most of the other direct approaches, hand-crafted linguistic rules and heuristics are employed. Each of these approaches has advantages and drawbacks that we have previously discussed.

Ghosh et al. [37] design a linguistic-based methodology called ARSENAL for specifying formal requirements from natural language. Acknowledging the fact that requirements engineers will prefer to specify requirements in unrestricted NL, the authors propose a

rules-based NLP pipeline that partially automates the formalization of such requirements. One of the first steps in the pipeline is to distinguish between domain-specific terms (e.g., 'lower desired temperature' in an industrial domain) and domain-independent terms (e.g., arithmetic expressions). This will be an important consideration for our system as well, since it is likely that text in a NL contract will correspond to objects in the Symboleo domain model.

The pipeline also uses a word standardization mechanism (gazetteer) to resolve synonymous expressions of a concept into a common term. This is a technique we saw applied to other formalization approaches, and it brings up the NL expressivity challenge that there may be *many* ways to state a given concept in NL. This inherent incompleteness is noted as a challenge by the authors. The proposed methodology is evaluated against a set of requirements from different domains.

A final piece of related work we discuss is Blawx [10], a web application aimed at policy makers that allows one to specify definitions, instances, and rules using a Blockly [11] interface. This type of interface constrains the user input to allow for predictable formalization of the rules, much like a CNL. Specific tests and queries can be written against the rules using the same type of interface to verify rule behaviour, and it makes use of Answer Set Programming techniques to generate explainable traces for answers for the queries.

3.6 Summary

This chapter began with a discussion of formal representations for legal documents and more specifically smart contracts, allowing us to understand the motivation behind the development of Symboleo and where it fits in to the specification language space. We then explored design considerations for CNLs, particularly those designed for normative documents. The PENS scale frames NLs, CNLs, and also formal representations as being different points on a complex scale, rather than completely different types of entities. NLs like English may be imprecise, expressive, and complex. Many formal representations are precise, inexpressive, and potentially simple. Many CNLs are between these two extremes, attempting to achieve a desired level of precision and expressiveness, while maintaining other metrics at desirable levels. Of particular importance in the design of our CNL will be the specification of events, and we discussed a few major initiatives that specify events as consisting of a set of frame elements (or slots), some of which may be considered more important than others. The principles behind the design of CNLs and the specification of events are rooted in existing linguistic constructs. Related work on event specification often involves restricting their structure, for example by only allowing the active voice or disallowing the nominalization of verbs. We will adopt some of these guidelines in our own event specification.

We then turned to methods of formalizing normative NL documents. Many linguistics-based approaches use techniques such as trigger phrase identification and pattern-matching that involves analyzing linguistic properties of the input text. These linguistic properties

[10] `https://dev.blawx.com/`

[11] `https://developers.google.com/blockly`

are often identified using a ML model that has been trained on a corpus of annotated text, and transferred to a new domain. While ML models can be useful, there is potential of carrying over bias from the training set. The linguistics-based approaches suggest heuristics that may incorporated into our CNL construction. Our goal of maintaining expressiveness means our CNL may not be completely precise and will thus require a certain degree of more probabilistic processing. Finally, we turned to work specifically related to the formalization of contracts that involved the use of a CNL or used SYMBOLEO as a representation. Our approach is strongly related to both of the ContracT [84] and the DSL4SC [88] initiatives, and can potentially make use of techniques seen in these and other approaches that were discussed. We are now ready to begin the construction of our CNL for contract refinements.

Chapter 4

Constructing a CNL

In this chapter, we will answer and evaluate our first three RQs, which deal with the construction of our CNL used for customizing a contract template. This CNL will be used in the implementation of SymboleoNLP, which is addressed in the subsequent chapter. A preliminary version of this construction process appears in Meloche et al. [62]. Before discussing the RQs, we will provide more detail on the contract template approach that we are taking in order to more precisely frame the RQs (Section 4.1). Our goal is to create a CNL that maps a NL contract refinement into a Symboleo operation. The first RQ will deal with defining the range of the output, or the allowed Symboleo operations for our CNL (Section 4.2). RQ2 will involve performing linguistic analysis to discover NL patterns that correspond to each of these operations (Section 4.3). RQ3 will involve applying these initial results to a dataset of real contracts in order to have a more empirical basis for our CNL (Section 4.4). We will conclude the chapter with a qualitative and quantitative evaluation of our CNL and the CNL construction methodology (Section 4.5).

4.1 Templating Approach

We have established that we want to use a template-driven approach for our customization. Here we will more concretely define the characteristics of these templates, leading to a more precise formulation of our problem statement and research questions.

4.1.1 Comparison with FITB

The simplest type of template for customizing contracts is a FITB template, where each blank has a very predictable data type. For example, consider the sentence: 'The seller shall deliver the goods to the buyer by [DELIVERY_DATE]'. The parameter here is DELIVERY_DATE and it would have the predictable data type of a date. We can imagine a simple contract customization application with a graphic user interface that would allow a user to select a date from a datepicker to customize this obligation. The formal specification for this obligation in Symboleo might be as follows: `O(seller, buyer, true, ShappensBefore(evt_delivery, DELIVERY_DATE)`

When the user fills this out, all the application would have to do is create a contract specification parameter (CSP) called DELIVERY_DATE with the date value selected by the user. The automation of this formalization is trivial, but it is not very customizable for a contract author. They can *only* enter a date. They do not, for example, have the option of specifying that the delivery must happen before another related event, or *between* two different dates. These are some examples of the customizability/flexibility that we can potentially integrate into our system. It is worth pointing out that the simple FITB approach does indeed have value, as there may be many contracts that are designed to be strict rather than customizable, and a simple automation tool would be all that is needed.

The FITB case is low-complexity and low-flexibility. Now consider an example where the user is given much *more* flexibility in the customization of the obligation: ROLE must PERFORM_ACTION QUALIFIER. This template gives the contract author lots of flexibility in how the obligation can be customized, but the formalization will be much more complex. We would need a tool that validates the ROLE and ensures that PERFORM_ACTION and QUALIFIER are both syntactically and semantically correct. The most extreme case of flexibility is full formalization, where there is no template at all. The user would enter freeform text, and the tool would formalize it into a Symboleo specification. While work towards full formalization is indeed a motivating factor of this work, we are restricting ourselves to a template-based approach, which involves fixing certain information that is essential to the contract. This also maintains the re-usability value that comes with using templates.

We therefore have a few important factors to consider in designing our templates. We want to push the boundaries on the flexibility of the input, but we must keep the template reasonably constrained for the sake of the end-user who is customizing the template. As the flexibility increases, the formalization becomes more probabilistic and therefore less certain, which means less value for the end-user. These factors are summarized on the spectrum in Fig 4.1.

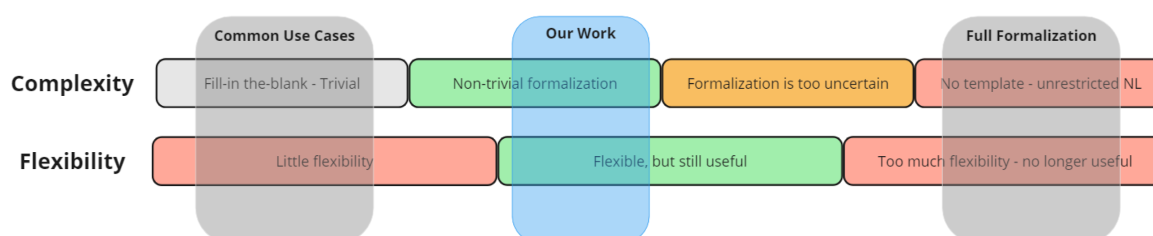


Figure 4.1: Spectrum showing the relationship between complexity and flexibility, as well as where the present work is situated.

4.1.2 Framing the RQs

Our NL contract template will consist of NL text containing parameters that can be customized by the user, which will go beyond simple FITB parameters. The crucial question to answer now is the precise linguistic forms that these parameters can take. The

answer will come in the form of a carefully designed CNL, and this is the main focus of our first three RQs. The end goal of our system is to convert a Symboleo template S(T) to a refined template S(C) based on the customizations. This involves a set of discrete operations applied to S(T). The goal of RQ1 is to decide which of these operations are desirable to include based on syntactic analysis of Symboleo.

If our proposed system were to deal purely with the Symboleo specification language, then customizations would be straightforward: We would present the user with the possible operations from RQ1, have them fill out any required arguments, and generate our refined specification. Both the input and output would be specified in Symboleo. However, we are ultimately dealing with human users as contract authors, who are unlikely to be familiar with formal specification languages such as Symboleo. Humans communicate using NL, which, while expressive and flexible, does not readily welcome formalization (recall the PENS classification). Furthermore, Symboleo is built on an ontological foundation which establishes relationships between the different contract and domain components. A simple change to the NL contract will likely result in more than one change to the specification, and these changes must be carefully controlled. Since we don't want to force users to learn a language they are unfamiliar with, we must adapt our proposed SymboleoNLP tool to receive something closer to NL as input, and then map that input to a formal specification. Therefore our next goal will be to discover the NL semantics that correspond to these Symboleo operations. These semantics will come in the form of linguistic *patterns*, which will result in a working CNL. This is the goal of RQ2.

This working CNL will be based purely on general linguistic analysis. We ultimately want a CNL that is tailored specifically to *legal contracts*. We will therefore frame the results of RQ2 as a set of linguistic *heuristics*. In RQ3, we will then apply these heuristics to a set of *real* contracts and identify linguistic patterns that correspond to our selected Symboleo operations, resulting in a final evidence-based CNL. We will then perform an evaluation on our final CNL to measure how well it covers contract refinements.

This CNL will become the basis of our tool, SymboleoNLP. With this tool, a contract author will fill out contract template parameters using the CNL, and the refinements will be automatically mapped to Symboleo operations, resulting in an updated specification of our contract, S(T). Since we are adding specific constraints to the structure of our parameters, we won't be able to simply start with any arbitrary contract template. Any contract template that we work with will first need to conform to the constraints that we will introduce. Converting an arbitrary contract template to meet these constraints will add another layer of manual work. We will address this in Chapter 5, after we have constructed our CNL.

4.2 RQ1: Choosing Symboleo Operations

The ultimate goal of the system we are designing is to transform one Symboleo formalization S(T) of a contract template T into *another* Symboleo formalization S(C), as illustrated in Fig. 1.1. This transformation involves a set of concrete operations C, so we must first understand the different types of operations that can be done to a Symboleo

document. For the time being, let us ignore any potential semantic correspondence between SYMBOLEO and NL and focus purely on the *syntax* of SYMBOLEO. To keep us grounded in the syntax, we will use generic identifiers for the arguments in subsequent examples. Consider the following two SYMBOLEO obligations:

- `ob1: O(partyA, partyB, true, Happens(eventX))`
- `ob2: Happens(eventY) -> O(partyA, partyB, true, WhappensBefore(eventX, Date.add(eventZ, 10, days))`

What are the concrete steps (operations) that will take us from ob1 to ob2? First, we consider the trigger. The obligation ob1 has no trigger, meaning it is created upon contract activation. Obligation ob2 has a trigger denoting that eventY must first happen in order to create the obligation. Therefore we could say that one operation would be to *add a trigger*. Another operation is to change the `Happens` predicate in ob1 to the `WhappensBefore` predicate in ob2. This would also need to be accompanied by the additional argument required for the `WhappensBefore` predicate.

These are just a few examples. Our goal is to list *all* possible operations that could be done to a SYMBOLEO specification, and then decide, using a set of guiding principles, which are desirable to include in our automated formalization system. In order to exhaustively list these operations, we must consider the various concepts in a SYMBOLEO contract (introduced in Section 2.1) and how each of them can mutate. We will then come up with a more formal desirability criteria for these operations to filter this list down.

4.2.1 Listing Potential Operations

We begin by considering the two main components in a SYMBOLEO contract: The domain model and the contract specification. The domain model is concerned with class definitions that extend ontology concepts, such as assets, events, and roles. The contract specification is concerned with the powers and obligations, which are built from declarations (instantiations) of domain model classes. The roles, assets, and events of the domain model all adhere to a similar structure. They have a string identifier and set of properties. Each property for a domain model class can be thought of as a key-value pair with a data type. For example, the Paid event has a property called amount of the predefined type Number (as well as other properties). A value for the amount will be specified in the declaration of an instance of a Paid event:

- **Domain Event:** `Paid isAn Event with amount: Number, currency: Currency, from: Role, to: Role;`
- **Declaration:** `evt_paid: Paid with amount := 100, currency := CAD, from := buyer, to := seller;`

At a high level, we can consider the operations of adding a completely new domain model class (role/asset/event) or deleting an existing domain model class. At a finer

level, we can look at the possible operations *within* a domain model class, namely on the properties, which we can add, delete, or modify. Adding/deleting a property simply involves adding/removing a new key-value pair to the class's property list. For example, on our payment event, we may want to add a property of payment_method, and allow possible values such as 'credit card', 'cash', and 'e-transfer'. Finally, since declarations are *instances* of these domain classes, we could consider the addition/deletion of a declaration to be another operation. The list of potential operations for the domain model (related to definitions and its usage in declarations) are as follows:

- Adding a class (role/model/asset);
- Deleting a class;
- Adding a property to a class;
- Deleting a property from a class;
- Declaring a new object (class instance);
- Removing an object declaration.

We now consider the operations applicable to the contract specification, which contains the norms (i.e., power and obligation legal positions). We first note the high-level operations of adding a brand new norm or completely deleting an existing norm. Going deeper, we consider the norm components. A norm contains 5 components (debtor, creditor, trigger, antecedent, consequent), so we consider the possible operations on each. The debtor and creditor are non-nullable references to role declarations, so we have the operation of modifying the value to reference another role (e.g., changing the debtor from 'partyA' to 'partyB'). The trigger, the antecedent, and the consequent all take the form of a Proposition, per our XText grammar specification (Listing 2.2). The trigger may initially be empty, meaning the norm is created at contract activation. The antecedent may initially be marked as 'true', meaning the norm is unconditionally in effect. For both of these concepts, we could add a more complete proposition. The consequent, however, will *always* contain an initial proposition, so it can only be *modified*. We list the possible operations to the contract specification:

- Adding a new norm;
- Deleting an existing norm;
- Modifying the debtor/creditor;
- Adding a trigger/antecedent;
- Modifying a trigger/antecedent/consequent.

The most interesting of these operations is modifying one of the propositions. We previously showed a sample of the syntax of a proposition in Listing 2.2, and saw that it can take a wide variety of forms. There are accordingly many ways of *modifying* a

proposition. We will limit ourselves to cases where the propositions take the form of predicate functions, since this is the case for the norms in many of the contracts that we have analyzed. A proposition can be composed of multiple predicate functions in conjunction or disjunction. The first way to modify a proposition could be to add or remove a predicate. Symboleo propositions also include the negation operator, so we have the possibility of reversing the negative polarity of one of the predicates in the proposition.

Finally, we narrow in on the operations that can be performed on a *single* predicate function. A predicate function is of a certain type (`Happens`, `WhappensBefore`, `HappensWithin`, etc.), and it contains a set of parameters. For a given predicate, we can therefore change the type (e.g., `Happens` → `WhappensBefore`) or we can change the arguments. If we change the type to one that has more arguments than the original type, then we also need to add these arguments in as part of the operation. We will see examples of this below. This notion of predicate types and arguments does not apply to the consequents of *powers*, which have a unique signature. Since powers are always specified in relation to *other norms or the contract itself*, there is a more precise set of forms that the consequent of a power can take, and are captured by the term *power function*. For example, one possible power function is the termination of an existing obligation. Since these power functions take pre-determined forms, we can omit them from our discussion for now.

4.2.2 Principles of Exclusion

Now that we have listed the potential operations that can be done to a Symboleo contract, we need to decide *which* of these operations we want our system to handle. It is important to point out that this question is mainly about *desirability*, rather than *feasibility*. Some operations may turn out to be too complex to integrate into the system when actually implemented. This, however, is not a reason to exclude them at this point. If high complexity ends up being the case for a given operation, then it can be included in a future roadmap for the tool. The goal here is to decide which operations are *desirable* for an ideal system. Feasibility will be addressed separately.

We have stated that a main goal for this research is to push the boundaries of flexibility. Therefore, we want to include every operation we can unless there is good reason to exclude it. As we have mentioned, if a template is *too* flexible, then the purpose of using a template for re-usability is defeated. To this end, we introduce some high-level restrictions on the operations that we allow. First of all, we want to keep the contracting parties (roles) intact, wherever they appear in the contract. Next, we want to keep the basic predicate of the consequent intact, which specifies how an obligation or power is fulfilled. Fixing these values maintains the essential information in the contract of who is obligated to do what. This entails the *dis-allowance* of the following operations:

- Adding new roles to the domain model;
- Removing existing roles from the domain model;

- Modifying the roles of a norm;
- Adding new norms to the contract specification;
- Removing norms from the contract specification.

Recall that the contract specification also includes concepts of constraints, pre-conditions, and post-conditions. Constraints typically encode assumptions about contract functionality in order to ensure properties like safety and liveness. Since these are typically implicit, we will exclude these from our system. The pre-conditions and post-conditions deal with assignability and ownership. Since we are fixing the roles, these are unlikely to change. This allows to focus primarily on the domain model classes and the content of the norms, but we acknowledge it as a limitation. Future work may involve gradually reducing some of the limitations introduced in the construction process.

We also want to adhere to the general principle that the purpose of using a template is to *refine* a contract. Even on a simple FITB contract, the customization always gives *more* information, making the result *more specific* than the template. We want to preserve this principle, and this results in further restrictions. This means we must disallow any remaining *removal* operations, specifically the removal of assets and events on the domain model. Removing information will generally broaden a contract rather than refine it. With these restrictions, we list the operations that remain:

- Adding a new domain model property;
- Adding a new domain model event;
- Adding a new domain model asset;
- Adding a trigger/antecedent to a norm;
- Declaring a new object (asset or event);
- Modifying a norm's trigger/antecedent/consequent.

We have four explicit 'adding' operations. The notion that these will refine the contract is fairly straightforward – they generally only ever add new information. From a purely syntactic point of view, we consider this to be a refinement. The addition of a new domain class will necessarily be accompanied by the *declaration* (i.e., instantiation) of the class as an object. It is also possible to create a new declaration from an *existing* asset or event from the domain model.

As for the final operation of modifying a proposition, we have acknowledged that there are many ways of doing so, and so we must consider these in greater detail to verify which of these ways align with our principle of only allowing restrictions. If the particular type of modification results in broadening rather than restricting the contract, we disallow it. Consider the simple obligation: `O(partyA, partyB, true, Happens(eventX))`. The types of modifications that can be done to a proposition are shown below, along with an example of how that modification would alter this obligation:

- Adding a conjunct predicate (AND)
  - Added to consequent:
    `O(partyA, partyB, true, Happens(eventX) AND Happens(eventY))`
  - Added to trigger:
    `Happens(eventY) -> O(partyA, partyB, true, Happens(eventX))`
  - Added to antecedent:
    `O(partyA, partyB, Happens(eventY), Happens(eventX))`
- Adding a disjunct predicate (OR)
  - `O(partyA, partyB, true, Happens(eventX) OR Happens(eventY))`
- Flipping the negative polarity of an existing predicate (NOT)
  - `O(partyA, partyB, true, not Happens(eventX))`
- Modifying the *type* of a given predicate
  - `O(partyA, partyB, true, WhappensBefore(eventX, dateD))`
  - `O(partyA, partyB, true, Occurs(situationS))`
- Modifying the *arguments* of a given predicate. This also may be entailed by modifying the *type*.
  - `O(partyA, partyB, true, Happens(eventY))`

The first operation of adding a conjunct also generalizes the case of adding the first predicate to the trigger or antecedent when it contains no predicates (i.e., simply the 'true' statement). This corresponds to a refinement and we therefore admit all three cases into our desired operations list. Adding a disjunct, however, broadens the scope, and so we exclude it. To highlight this, consider the statement 'You must wash the car'. Introducing a conjunct ('You must wash the car *and* mow the lawn') is more specific than the initial statement, while introducing a disjunct ('You must wash the car OR mow the lawn') is less specific. Flipping the negative polarity is neither a broadening nor a refining operation – it could be considered neutral. Since our principle is to include only refining operations, we also exclude it.

4.2.3 Refining and Broadening Predicates

For the next two operations (modifying the predicate type and the predicate arguments), we first review the list of our SYMBOLEO predicates from Table 2.2. Reminding ourselves that we are currently focused primarily on a *syntactic* perspective, a modification of a predicate type could only be considered a refinement if the new type contains *at least* the parameters of the original type. For example, if we were to modify `Happens(eventX)` to `WhappensBefore(eventX, dateD)`, this involves purely adding information and would therefore be considered a refinement (assuming we do not alter the eventX argument). If we

were to go the other way (`WhappensBefore` → `Happens`), then this would be considered *broadening* the predicate, since we are removing information. If we were to take two predicates with completely different signatures (e.g., `Happens` → `Occurs`), then this would *not* be considered a refinement. Following our principle of refinement, we admit only the operations where the new predicate contains at least all of the arguments of the old predicate. We can exhaustively list these operations:

- `Happens` → `WhappensBefore`
- `Happens` → `ShappensBefore`
- `Happens` → `WhappensBeforeEvent`
- `Happens` → `ShappensBeforeEvent`
- `Happens` → `HappensAfter`
- `Happens` → `HappensWithin`

This section is focused on a syntactic perspective, but briefly considering these refinements from a *semantic* perspective may be illustrative. Compare the sentence 'you must eat breakfast' with 'you must eat breakfast before 9AM'. The second is a refinement of the first, in that it adds more information and thus more restrictions (what is valid for the refinement is also valid for the initial sentence). We will deal more with the semantic perspective in the next section, including a discussion on the distinction between `ShappensBefore` and `WhappensBefore`.

We have now defined a set of Symboleo operations that are within the scope of a template-driven approach, in that they prevent excessive flexibility, they maintain essential contract information, and they only allow refinements on the contract. Our next step will be to consider the semantics of these chosen operations, and match them to corresponding linguistic structures. These structures will create a basis for the CNL patterns that the user will be allowed to enter into the parameters of a contract template. We note that the process of semantic analysis may also involve further restrictions on the operations. We also note that the restrictions that we have introduced will result in limitations with what can be expressed by our CNL. Relaxing some of these restrictions to cover more use cases is a possibility for future work.

Another important result of this investigation is that we have begun to concretely define what is meant by *refining* in terms of Symboleo syntax. A norm is considered *broad* if it has concepts that can be refined, such as an empty trigger, or a consequent that uses the Happens predicate. The first can be refined through the addition of a predicate, and the second can be refined by modifying the `Happens` predicate to a `WhappensBefore` predicate, for example. This has important implications for how our task will be framed. If the allowed operations consist of going from broader norms to more refined norms, then it follows that the most flexible Symboleo template will be one which is *maximally broad* – In other words, a template where obligations have *only* empty triggers, true antecedents, and all consequent predicates of the type `Happens` or `Occurs`, for example: `O(partyA,`

`partyB, true, Happens(eventX))`. If we have such a contract template, then there are many ways in which it could be refined, making it *maximally flexible*. The `Occurs` predicate does not appear in our list of predicate refinement operations, because there is no way to syntactically *refine* it into another predicate. It can appear in a maximally broad contract, however, because there is no way to syntactically *broaden* it.

We emphasize that this RQ has been concerned with *syntactic* analysis, and that *semantic* analysis may motivate us to relax or tighten some of the restrictions we have introduced. To transition to our discussion of RQ2, which deals with the semantic interpretation of these operations, we summarize our allowed operations with an example of each one.

- **Adding a domain event**
  Suppose we have a property rental contract between a renter and landlord. We want to specify that a certain obligation for a renter is violated in the event that the renter abandons the property. We would therefore need to add a domain event concerned with 'Abandoning' the property to the domain model, as well as a corresponding declaration.

- **Adding a domain asset**
  If we're specifying an obligation on a chemical equipment delivery contract that involves verifying the quality of lab test results, then we may need to add 'LabTestResults' as an asset to the domain model, as well as a corresponding declaration.

- **Adding a predicate to a proposition**
  Suppose we changed *If the seller has not delivered the goods* to *If the seller has not delivered the goods AND they have failed to notify the buyer of the reason.* This operation also covers cases where *no* predicate exists for the proposition, and we are adding the *first* one.

- **Declaring a new domain class instance (declaration)**
  Suppose we have an existing domain event that specifies a 'Delivery' event. If we want to add a separate instance of delivery (for example, with the roles reversed, or with a different item to be delivered), then we would need a new declaration.

- **Refining a Happens predicate**
  Specifying that a delivery must happen *before* a certain date would require us to refine the predicate function in the consequent from `Happens` to `WhappensBefore` and add the date as an additional argument to the predicate.

4.3 RQ2: Finding NL Correspondence

Now that we have a set of concrete SYMBOLEO operations, we can analyze the NL semantics that correspond to each of these operations. The linguistic patterns we identify for our operations will form the basis of our CNL. We've discussed definitions, strategies, and various criteria for characterizing a CNL. With those discussions in mind, we will now highlight some key motivations and considerations involved with CNL design with respect to a formal language like SYMBOLEO.

4.3.1 CNL Design

Since NL is expressive, there are multiple ways of stating the same idea. Consider our earlier refinement example of `'within 2 weeks of payment'`. We could conceive of many other synonymous NL phrases, for example `'before 2 weeks have passed since payment'`, or `'by no later than 2 weeks following payment'`. All of these would likely correspond to the same refinement operation in Symboleo. In general, we can say that no Symboleo operation will have a unique correspondence in NL. This presents a problem since, in order to consider our system *complete* in a certain sense, we would want to find *all* possible NL correspondences to a given Symboleo operation and make sure we program the mapping in. If a user enters a valid NL phrase that they expect to result in a certain type of smart contract behaviour, then ideally that should be programmed into the system. However, it is impossible to anticipate every possible NL phrase that would correspond to a given Symboleo operation. There will therefore be situations where a user may *wish* to enter one way of expressing a certain idea, but the system is not equipped to handle it.

Conversely, for many NL phrases, there may be multiple interpretations about the correct way to map it to Symboleo. This would be the case for distinguishing between a trigger and an antecedent, for example. While there is a clear distinction in terms of the precise semantics that correspond to the state machine of Symboleo, both are conditional in the sense that they render an obligation active. This is not so much an issue for the user, since they need only be concerned with the NL interface rather than the details of Symboleo, but the problem still exists for us in deciding how to program the mapping in the application.

We therefore have two problems of ambiguity. First, one Symboleo operation may have multiple corresponding NL phrases. Second, one NL phrase may correspond to multiple Symboleo operations. Both problems can be mitigated by the introduction of a CNL. A CNL would use a small subset of NL words, word types, and grammar arranged into pre-defined patterns, each of which having a predictable mapping to a corresponding Symboleo operation. Since the user would only be able to enter text that corresponds to our set of patterns, we mitigate errors arising from unanticipated NL phrases (first problem), and our pre-defined mappings will enforce that each NL input maps reliably to a single Symboleo operation (second problem). We are now left with the problems of deciding *which* words/patterns to include in the CNL and deciding which Symboleo operations the patterns will map to.

While a CNL addresses the ambiguity problems, it is not without drawbacks. We have noted that the introduction of any restrictions on the user input will reduce the expressiveness inherent in NL. There will be situations where the user may *wish* to enter some grammatically correct text, but it is unsupported by our system, as they can *only* enter input in the form of patterns that we define. However, by following proper software development practices, we can ensure that our system is extensible and that it is relatively easy to integrate new NL patterns as the need arises. We also emphasize that we are restricting the NL to *patterns*, rather than a strict set of *words*. This gives the user some degree of flexibility in entering in unanticipated events and actions, so we can define our patterns to allow for a limited amount of freeform input. This has the potential

to introduce a degree of uncertainty in the processing, as there is a trade-off between reliability and flexibility. This trade-off is closely related to the trade-off between precision and expressiveness found in the PENS classification schema.

Finally we note that since this is not completely unrestricted NL, there will be a learning curve for a user; The user will need to know which phrases and words are valid. This relates to the simplicity factor of the PENS schema. The barrier to learning a subset of NL, however, will be much lower compared to learning a new formal language. Furthermore, this learning curve can be mitigated by building the tool in such a way that the user is guided through a process where they are *only* able to enter valid patterns. This might be in the form of an auto-complete or drag-and-drop mechanism. While we want to incorporate the structured patterns into our system, optimizing the specific UI design is not a focus of this work. We will discuss mechanisms for generating and enforcing valid user input in Sec. 5.2.

4.3.2 Principle of Integrity and the Adjunct

We will now introduce an important assumption that will be made by our system: the *principle of integrity*. This principle states that at any point in the customization process, we will always maintain a valid SYMBOLEO contract. Another way of saying this is that we want our NL parameters to be *optional*. If our template contains a number of parameters for the user to fill, and they only fill one of them, we still want to yield both a grammatically correct NL contract and also a valid SYMBOLEO specification. Since the operations we support only serve to *refine* a contract, any unfilled parameters will simply result in a relatively *broad* contract. We want to minimize assumptions about the user's end goal, and so we admit the possibility of contracts where broadness may be desired.

A consequence of this principle is that the entire original NL contract template T must be grammatically correct and the corresponding specification S(T) must be valid SYMBOLEO. As an example, consider the parameterized obligation: 'Party A must pay party B $100 [PAYMENT_REFINEMENT].' This could be customized to the following: 'Party A must pay party B $100 *before April 18, 2024*'. The parameter is customized to `'before April 18, 2024'`, but the user could enter a variety of other NL phrases, such as `'within 2 weeks of delivery'`, `'unless party B forgives the debt'`, etc. The point is that the template is still grammatically correct, even if the PAYMENT_REFINEMENT parameter is completely ignored. We would simply end up with the text 'Party A must pay party B $100'. This sentence is grammatically correct and it produces a valid, albeit broad, SYMBOLEO obligation: `O(party A, party B, true, Happens(evt_payment))`. However, if the user customizes the PAYMENT_REFINEMENT parameter to `'before April 18, 2024'`, then the obligation is refined as follows: `O(party A, party B, true, ShappensBefore(payment, payment_date))`. The `payment_date` variable with the value 'April 18, 2024' would be passed in as an argument to the contract instantiation.

Other NL patterns entered by the user will map to different SYMBOLEO operations defined in RQ1. This is a single example, but we want to know what the *general* patterns of our NL parameters will be. Our two requirements for these patterns are that they

must have semantic correspondence with our desired Symboleo operations and that they must be optional. A linguistic structure of interest is therefore the *adjunct*, which is a structurally dispensable (i.e., optional) constituent that *refines* a sentence. While the adjunct is somewhat open-ended and context-dependent, it provides us with a starting point in finding semantic correspondences with our Symboleo operations.

4.3.3 Specifying Building Blocks

With the establishment of the principle of integrity and a close correspondence between our requirements and the linguistic adjunct, we are now closer to specifying the semantics of each Symboleo operation from RQ1. The general strategy that we take to find a semantic correspondence for a given Symboleo operation will be as follows:

1. Re-iterate the more formal semantic meaning of the operation.
2. Note any obvious or trivial semantic alignment, and provide example use cases in NL and the corresponding Symboleo.
3. Use these trivial NL correspondences as a starting point for finding synonymous ways of representing the operation. We can make use of various lexical resources, such as thesauri or advanced language models like ChatGPT [1] that can be prompted, to find synonyms.
4. Identify broader linguistic patterns from these NL representations, and incorporate these patterns into a grammar.

The result will therefore be a set of broad patterns that correspond to each Symboleo operation from RQ1, as well as an accompanying grammar for these patterns. Many of the Symboleo constructs make use of some basic building blocks, namely the event, timepoint, interval, and situation (Fig. 2.1). Since these will be used at various points throughout the following sections, we start our grammar construction by first discussing how some of these building blocks can be specified in NL.

Events

A common form for an adjunct to take is a prepositional phrase with an event clause as the complement. Examples include (with the event clause in italics):

- Before *payment is completed*
- After *the contract terminates*
- If *legal proceedings become necessary*

[1] `https://chat.openai.com/`

There is no limit to how events can be expressed in NL, but since we are constructing a CNL, we can restrict the way that events can be specified. We re-iterate that any restrictions introduced in a CNL will reduce the expressivity, so we want to make sure the formats we admit are sufficiently expressive, natural, and familiar. We introduce the notion of a *standard event* in Listing 4.1, which is the event format that we will be dealing with throughout this work. This specification is based largely on ideas from preliminary linguistics principles, related work surrounding the specification and framing of events (FrameNet, Attempto Controlled English (ACE)) as well as recommendations from the CNLs we studied in Section 3.2.

Listing 4.1: Grammar for specifying a standard event

```
<EVENT> ::= <SUBJ> <VP>
<SUBJ> ::= simple_np
<VP> ::= [<ADVB>] (<TVP> | <LVP> | <IVP>) [<PP>]
<ADVB> ::= adverb
<TVP> ::= transitive_verb <DOBJ>
<DOBJ> ::= simple_np
<LVP> ::= linking_verb <PRED>
<PRED> ::= adjective
<IVP> ::= intransitive_verb
<PP> ::= preposition <POBJ>
<POBJ>::= simple_np
```

A standard event is a declarative clause consisting of a subject and a verb phrase. The verb phrase may be preceded by an adverb and may be followed by up a prepositional phrase (PP). The verb itself may be transitive, intransitive, or linking, and this verb type determines what can follow the verb. Concepts such as simple_np, adjective, and the verb types require further elaboration. For example, the possible values for a simple_np might be the declared roles and assets in a contract, or a noun phrase containing an article and a few adjectives. These parts of speech may be identified by NLP tools such as SpaCy, and we will discuss them in more detail later on. Examples of events that would meet the grammar listed above include:

- The seller delivers the goods to the buyer (transitive VP)
- Legal proceedings become necessary (linking VP)
- The rental agreement terminates (intransitive VP)
- Bob noisily eats apple pie with Alice (transitive VP)

The grammar shown above can capture many relevant event expressions, but it is still quite restrictive with respect to the entire English language. The refinements that we see in real contracts will certainly be more complex than these standard events. These more complex events will not be able to be represented in our SymboleoNLP tool, which we acknowledge as a limitation. We note, however, that as this system evolves, the grammar of an event can be made more sophisticated to allow for more complexity. Concepts to add in that would make the event specification more expressive include negations, conjunctions,

and subordinate clauses. This would require more complex grammars, which we will not treat here.

For now, we abstract away the notion of an event into the EVENT rule and note the hidden complexity. We also note that the event clause may take on various inflections, depending on how it is used in a full sentence. For example, if our event is 'the contract terminates', it may be used in this present tense (as in the example 'before the contract terminates') or in the continuous tense (as in the example 'within 2 weeks of the contract terminating'). We include allowance for these formats and inflections within our abstraction of the EVENT rule.

Timepoints

A timepoint may refer to a specific date (e.g., March 18, 2024), the time at which an event happens (e.g., payment being completed), or a time that is *relative* to an event (e.g., 3 weeks after payment is completed). We will refer to this relative time as a *point function*. We will discuss all three characterizations of a timepoint.

A date specification is straightforward, but can come in many formats. Certain components of the date, such as the month, may be spelled out (January 17th, 2024), it may be abbreviated (Jan 17, 2024), or it may use slashes (2024/01/17). If using a slash-based notation, it is important that the user knows which specific format is being used (e.g., YYYY/MM/DD, MM/DD/YYYY, etc.). We will capture all such formats under the DATE rule.

A timepoint can also take the form of an event, in which case we can simply use our EVENT rule previously discussed. Finally, a timepoint can take the form of a point function, which refers to a time *relative* to some other event. This characterization *includes* an event expression, so we can once again reuse our EVENT rule, but for the *relativity* time aspect, we introduce the TIMESPAN, which consists of a number followed by a time unit (e.g., hours, days, weeks, months, years, etc.). To complete the point function specification, we need to link the TIMESPAN with the EVENT using a temporal preposition ('after' in our example). Since we want to capture relative times both *before and after* an event, we want to consider any possible synonyms for before and after. Some examples include:

- **'before' keywords:** 'before', 'by', 'prior to', 'earlier than', 'preceding', 'in advance of', 'ahead of'
- **'after' keywords:** 'after', 'following', 'later than', 'once', 'upon'

Not all of these could be used directly in a point function as we have framed it. We specifically need prepositions that can be used when the complement of the preposition is an event clause. Recall that certain traditional grammars may not consider these to be prepositions at all, but rather subordinating conjunctions. However, we will consider these to be prepositions. With some of these prepositions, we also may need to inflect the verb in the event clause. We will capture the valid prepositions for a point function in the P_PF rule. The grammar for a timepoint is specified in Listing 4.2. Some examples of a point function could include:

- 3 weeks prior to the contract terminating
- 2 days after the buyer pays the seller
- 1 month ahead of the unit being leased

Listing 4.2: Grammar for specifying a timepoint

```
<TIMEPOINT> ::= <DATE> | <EVENT> | <POINT_FUNCTION>
<DATE> ::= *various date formats
<EVENT> ::= *see above specification discussion
<POINT_FUNCTION> ::= <TIMESPAN> <P_PF> <EVENT>
<TIMESPAN> ::= <TIME_VALUE> <TIME_UNIT>
<TIME_VALUE> ::= Number
<TIME_UNIT> ::= "hours" | "days" | "weeks" | ...
<P_PF> ::= "before" | "prior to" | "after" | ...
```

Intervals

A time interval consists of a beginning timepoint and an ending timepoint, which as noted, can take the form of a date, point function, or an event. An example of this is `'between March 31, 2024 and the contract termination'`. In this case the first timepoint is a date, and the second is an event statement. Intervals can also be framed in terms of time periods ('during the weekend'), using timespans ('for 6 months'), or by combining timespans with timepoints ('for 6 months following the next full moon'). We generalize the following patterns:

- between TIMEPOINT1 and TIMEPOINT2
- from TIMEPOINT1 until TIMEPOINT2
- during TIME_PERIOD
- for TIMESPAN following TIMEPOINT

There are many variations on these patterns that could also represent an interval, but these provide some useful keywords that can be used as a starting point. Ironically, the keyword 'within' is more suggestive of a `HappensBefore` refinement, though it could potentially be used for a `HappensWithin` refinement, albeit somewhat awkwardly. The notion of an interval becomes complicated when we consider the framing of statements as either events or situations. Here we are referring to the *linguistic* notion of situation, rather than a situation as specified by Symboleo. Consider again our 'disclosure' example:

- **Situation:** Party A must keep the document confidential
- **Event:** Party A must not disclose the document

Situations and events can each be refined in different ways using an interval. Used with an event E, we are saying that *E must take place* sometime between T1 and T2. Used with a situation S, we are saying that *S must hold* for the duration of T1 until T2. In SYMBOLEO, we can *only* use events, so we must use the negated event re-framing that was previously discussed. However, the NL expressed in the contract template may be framed as a situation or an event, and certain patterns we defined above will not necessarily make sense with events. For example, the statement 'Event E must happen from TIMEPOINT1 until TIMEPOINT2' makes no sense. The very usage of the words 'from' and 'until' suggest a *situation* rather than an *event*. This is also the case with For TIMESPAN following TIMEPOINT. However, if the event is has a negation, then these patterns *do* make sense:

- Event E must *not* happen from TIMEPOINT1 until TIMEPOINT2
- Event E must *not* happen for TIMESPAN following TIMEPOINT

For statements framed as situations, however, all of our interval specifications make sense (e.g., 'Situation S must hold for TIMESPAN following TIMEPOINT'). For now, we will limit our grammar to slightly more general versions of the patterns mentioned above, but we will note that some of these *cannot* be used if the statement is framed as a *non-negated event*. The interval specification is found in Listing 4.3.

Listing 4.3: Grammar for specifying an interval

```
<INTERVAL> ::=
    "between" <TIMEPOINT> "and" <TIMEPOINT>  |
    ["from" <TIMEPOINT>] "until" <TIMEPOINT> |
    "during" <TIME_PERIOD> |
    "for" <TIMESPAN> P_AFTER_I <TIMEPOINT>
<P_AFTER_I> ::= "following" | "after" | ...
<TIME_PERIOD> ::= *various specifications
```

Note that we've made the from TIMEPOINT optional in the definition, since it is possible to specify an interval using only 'until', in which case the initial timepoint is assumed to be the present time. The notion of TIME_PERIOD is currently open-ended and can include concepts like 'the weekend', 'work hours', 'the term of this agreement', etc.

Situations

We re-emphasize the distinction between a situation represented in NL (e.g., 'Party A must keep the documents confidential') and a situation in SYMBOLEO, which refers to a state specifically associated with a SYMBOLEO norm or the contract. Here we are referring to the latter. The SYMBOLEO concept of a situation appears only in the `Occurs` predicate, and since none of our refinement operations include this predicate, we will not be dealing with expressing SYMBOLEO situations in our CNL.

4.3.4 Predicate Refinements

With the relevant building blocks of our Symboleo constructs now specified in grammatical detail, we can now turn to the specific refinement operations outlined in RQ1. We begin with the set of predicate refinements, which are used for refining the temporal aspect of a norm, and will therefore be closely related to certain temporal adjuncts, which often take the form of PPs. Recalling the semantics listed for the predicates in Table 2.2, our goal is to establish a set of adjuncts for each predicate that correspond to those meanings.

Happens → HappensBefore

The Symboleo specification contains four predicates that are all closely related to each other in the sense that they represent an event happening *before* a *secondary timepoint*: `WhappensBefore, ShappensBefore, WhappensBeforeEvent, ShappensBeforeEvent`. As mentioned, a timepoint can take various concrete forms, such as an event, a date, or a point function. Since we have predicates that specifically refer to the 'event' characterization of the secondary timepoint (`WhappensBeforeEvent` and `ShappensBeforeEvent`), we will restrict this current discussion to `WhappensBefore` and `SHappensBefore` predicates and consider the secondary timepoint strictly as a date or a point function. The W and S in these predicates stand for 'weak' and 'strong' respectively. In `ShappensBefore`, there is a guarantee that the timepoint will eventually happen, which is not the case with a `WhappensBefore`. Given a generic NL statement that event `e` happens before timepoint `t`, how do we know that `t` will eventually happen? Consider two examples:

- 'The application must be submitted before Thursday, April 18, 2024.'
- 'The application must be submitted within 2 weeks of the deadline closing.'

If `t` takes the form of a date, as it does in the first example, then we *do* have a guarantee that `t` will pass (apocalypse aside, April 18, 2024 is *guaranteed* to happen), and we can safely map it to `ShappensBefore`. If, however, the second argument is an indefinite timepoint, we cannot be sure. In the second example, we have no guarantee that 'deadline closing' is an event that will happen. In cases like this, we will map to `WhappensBefore`, since it makes no assumption of the second argument coming to pass. We therefore restrict the `ShappensBefore` operation to correspond to NL expressions involving an event happening before a *specific date* (first example). We restrict the `WhappensBefore` operation to correspond to NL expressions involving a *point function* (second example). From a semantic perspective, there may be a case for including a refinement operation of strengthening a weak predicate into a strong predicate, for example by changing an event to a specific date. From a syntactic point of view, this would involve the replacement of a parameter. We generally disallowed this type of operation in RQ1, but future work may involve incorporating more complex refinements such as this. Let us now look closer at the `ShappensBefore` case, followed by the `WhappensBefore` case.

Suppose that our template contains NL corresponding to a simple `Happens` predicate (e.g., 'The application must be submitted [PARAMETER]'), and we want to list the

possible patterns that PARAMETER could take that would result in a correspondence to a ShappensBefore refinement. The obvious candidate for specifying that an event happens before any date is a PP headed by the preposition 'before', as we see in our first example. This can also be determined by considering the semantics described in Table 2.2. We can represent this basic pattern as before DATE. We can generalize this to P_BEFORE_S DATE, where P_BEFORE_S includes prepositions that we listed as synonyms for 'before' (e.g., 'prior to', 'earlier than', etc.). We will make a clear distinction between these two concepts. We will refer to patterns such as before DATE simply as *patterns*. A pattern will involve replacing the building blocks specified above with their rule names (EVENT, TIMESPAN, DATE, etc.). The rest of the NL will stay intact. We will refer to P_BEFORE_S DATE as a *pattern class*. In a *pattern class*, we will further generalize certain synonymous words (such as 'before', 'prior to') into rules (P_BEFORE_S). It is therefore possible for a pattern class to include multiple patterns.

We can now look at the WhappensBefore refinement, where the timepoint will take the form of a point function. Many of the prepositions used for the ShappensBefore refinement will not apply here, since certain prepositions are grammatically incompatible with a POINT_FUNCTION. We could potentially use the preposition 'before': 'before 2 weeks after the deadline closes'. While this is grammatical, the wording is a bit awkward. We can tentatively admit some of these patterns, with the knowledge that we may discard them if we find that they are not often found in real contracts (RQ3). We generalize the following pattern class for this operation as P_BEFORE_PF POINT_FUNCTION, and add the following specifications to our grammar, where P_BEFORE_PF includes prepositions such as 'before', 'prior to', 'earlier than', etc. We also note a special case of the 'within' preposition, exemplified by the refinement 'within 2 weeks of the contract terminating'. The 'of' following the TIMESPAN only applies if the head word is 'within'. For example, 'before 2 weeks of the contract terminating' is ungrammatical. We must capture this as a special case in our grammar. We can also replace the word 'of' with various synonyms for 'after', which capture in the P_AFTER_W rule, and introduce the pattern class within TIMESPAN P_AFTER_W EVENT.

Happens → WhappensBeforeEvent

We have two predicates that specify that some event (e1) happens before another event (e2): ShappensBeforeEvent and WhappensBeforeEvent. The semantic distinction between these predicates is the same as with our previous predicates; In ShappensBeforeEvent, e2 is guaranteed to happen eventually, and in WhappensBeforeEvent, it is not. We have discussed how the 'Strong' case should only be used for dates. Since these predicates refer strictly to events, which we cannot generally guarantee to happen, we argue that the 'Strong' case is not currently needed for our work, though there may be a use case for it in the future. The lack of a guaranteed completion time is also a noted practical drawback to Symboleo, and it is also possible that it may be removed or altered from future versions of Symboleo. For now, however, we discard the Happens → ShappensBeforeEvent operation and focus solely on the Happens → WhappensBeforeEvent operation. We can consider the simple example: 'The buyer must pay the seller before the contract terminates'. The obvious NL pattern for this operation is before EVENT, and we once again consider

synonyms for 'before'. Most of the prepositions used in the the previous refinements can be used, with some slight grammatical alterations of the event specification:

- 'The buyer must pay the seller *by the time* the contract terminates'.
- 'The buyer must pay the seller *prior to* the contract terminating'.
- 'The buyer must pay the seller *earlier than* the contract terminating'.
- 'The buyer must pay the seller *ahead of* the contract terminating'.

For most of these cases, the alteration involves using the continuous form of the verb (terminating). We could also potentially turn the event into a NP (contract termination), in which all of the prepositions would apply (e.g., 'The buyer must pay the seller *preceding* contract termination'). We note, however, that this was cautioned against in our study of existing and proposed CNLs [26]. All of these examples may be considered valid patterns for the `WhappensBeforeEvent` refinement, and we can generalize it to the pattern class P_BEFORE_E EVENT, where P_BEFORE_E represents valid prepositions discussed above.

Happens → HappensAfter

This refinement involves specifying that a given event happens *after* a timepoint. Note that no distinction between *strong* and *weak* predicates is required for this case, since by definition, the event in question will have *already happened*. We will again consider cases where the timepoint is a date, a point function, or an event. For the case where the timepoint is a date, we can consider the example 'Seller must deliver the goods after March 1, 2024': `O(seller, buyer, t, HappensAfter(evt_delivery, target_date))`, where the `target_date` refers to an argument passed in to the contract specification, with a value of 'March 1, 2024'.

This example highlights the obvious correspondence to a PP headed by 'after' which is used in a temporal sense. Once again, we consider synonyms and other ways of expressing the statement. Alternative ways of characterizing the PP in our example include:

- `'following March 1, 2024'`
- `'later than March 1, 2024'`
- `'once March 1, 2024 occurs'`
- `'upon March 1, 2024 occurring'`

The 'following' and 'later than' examples are fairly natural, but the others are somewhat awkward and require an additional verb ('occurs', 'occurring') to be grammatically correct. Since we are only concerned with expressing a date as a noun phrase, we will admit only the 'after', 'following' and 'later than' prepositions under the P_AFTER rule. The corresponding pattern class is therefore P_AFTER DATE.

We also want to consider cases where the timepoint may be expressed as a point function, as in the example 'Seller must deliver the goods after 2 weeks before the contract terminates'. While the phrasing again sounds awkward, it is still grammatical, and we therefore want to keep it under consideration. The pattern class will be P_AFTER_PF POINT_FUNCTION. P_AFTER_PF will contain prepositions synonymous with 'after' that license a POINT_FUNCTION. This includes 'after', 'following', and 'later than'. This pattern allows for some very unintuitive phrasing, for example: 'following 2 weeks prior to the contract terminating'. Once again, we note that some patterns may be filtered out when we analyze real contracts in RQ3.

Finally, we consider cases where the timepoint takes the form of an event, as in 'Seller must deliver the goods after the buyer completes the payment'. We could imagine a `HappensAfterEvent` predicate, but since the arguments for `WhappensBeforeEvent` are *already* two events, we can potentially use this same predicate to represent the 'Happens After Event' scenario. All we would need to do is swap the places of the event arguments: `WhappensBeforeEvent(evt_payment, evt_delivery)`.

Since we have an event as the complement of the PP, we can use a wider variety of synonyms for after, including: 'following', 'later than', 'once', and 'upon'. We will call this collection P_AFTER_E. These could be used as follows, where the event takes the form of a NP:

- '*following* payment by the buyer'
- '*later than* payment by the buyer'
- '*once* the buyer completes the payment'
- '*upon* payment by the buyer'

We therefore have the tentative pattern class P_AFTER_E EVENT, which will map to a `WHappensBeforeEvent` refinement, with the swapped argument order.

Happens → HappensWithin

This refinement involves specifying that an event happens within a certain time interval. An example of this is 'The seller must submit an invoice between March 31, 2024 and the contract termination'. We can also imagine more complex expressions that *break up* the statement of the interval: 'The seller must submit an invoice after March 31, 2024, and it must be submitted no later than contract termination'. This might suggest the creation of two separate obligations (one involving a `HappensAfter` and another involving a `WhappensBefore`). Indeed, this is essentially what a `HappensWithin` predicate refers to, in addition to an assumption that the first timepoint argument precedes the second. We will focus on NL patterns that contain an entire interval statement, and note this restriction as a limitation.

The linguistic complexity for the `HappensWithin` refinement is all captured within the INTERVAL specification described earlier. Therefore, we do not require any further

keywords to specify the patterns associated with this operation. The refinement simply consists of an INTERVAL following the statement of an event.

We've now established some general NL patterns that are semantically equivalent to our Symboleo operations that involve refining a predicate. We've focused largely on PPs beginning with certain prepositions, but there may be cases where other types of adjuncts or even other linguistic constructs represent a predicate refinement, perhaps through the use of figurative language. Therefore, we cannot say that our semantic characterization of these refinements is *exhaustive*. RQ3 will involve verifying if these pattern classes or variations of them exist in real contracts, and will provide us with a degree of concrete evidence about the most common forms of these refinement patterns.

4.3.5 Adding Predicates

Let us now consider the Symboleo operation of adding new predicates to an existing proposition. Recall that the trigger, antecedent, and consequent of a norm all take the form of a 'Proposition', which can be made up of a conjunction of predicates. We've discussed two scenarios where we could add a predicate: When there is no existing predicate in the proposition, and when a proposition already has at least one existing predicate.

In some cases, the existing proposition may simply be 'true', as in the case of an antecedent in an unconditional obligation. Or it may be *empty*, as is the case of a trigger on obligation that is created immediately upon contract activation. For the purposes of this discussion, both of these cases correspond to the first scenario. Since the consequent will always contain at least one non-empty predicate (different from 'true'), this scenario is applicable only to triggers and antecedents. Recalling our discussion on the distinction between triggers and antecedents, we can now ask what the NL semantics for each of these look like.

Triggers and Antecedents

If we have an adjunct that suggests a sort of frequency or a recurrence, then we can treat it as a trigger. This could be in the form of frequency adverbs, such as 'monthly' or 'weekly', or it could come in the form of conditional adjuncts that start with certain keywords, such as 'whenever', which has an implication that the event in question might occur multiple times. Other than these types of semantic heuristics, we don't have a clear way of distinguishing between these two concepts. With the standard 'If' statement, for example, the word 'If' does not necessarily imply recurrence. Therefore we cannot say that it should *for sure* be a trigger, but we also cannot rule a trigger out; There's no reason to prefer one interpretation over the other. For NL refinements that include the heuristic of an implied recurrence, we will treat them as triggers. For other conditional statements in general, we will treat them as antecedents. The current version of the Symboleo specification does not allow for recurring events. We will therefore exclude explicit adverbs and adjuncts of frequency from consideration. Since Symboleo is under active development, we note that this may change in the near future, at which point we can extend our CNL to accommodate. It may also be the case that in certain domains, specific

event types will always be classified as triggers or antecedents, which would motivate further semantic analysis.

We can summarize this by stating that adjuncts using 'when' or 'whenever' will map to a trigger, and all other conditional adjuncts will map to an antecedent. We will group the first set of words under the CONDITIONAL_T rule and the second group under CONDITIONAL_A.

Conditional Adjuncts

We illustrate the conditional adjunct with an example: 'If delivery is completed, the buyer must pay the seller'. The equivalent SYMBOLEO is: `O(buyer, seller, Happens(evt_delivery), Happens(evt_payment))`. Omitting the conditional adjunct results in 'The buyer must pay the seller' and corresponds to removing the antecedent: `O(buyer, seller, true, Happens(evt_payment))`. In keeping with our principle of integrity, the sentence is still grammatically correct after this removal, and the SYMBOLEO is valid. We therefore have alignment between the SYMBOLEO operation of adding an antecedent and the general conditional adjunct.

We can now consider other possible ways to express a conditional adjunct. The most intuitive way to represent a conditional in NL is simply to use an 'If' statement followed by an event specification, as we did in our example. We consider the pattern class CONDITIONAL_A EVENT, and list phrases that could be included in CONDITIONAL: 'If', 'In the case of', 'Upon', 'Assuming', 'Once', 'Provided that', 'On the condition that', 'Under the circumstance of', 'In the event of', 'Should', 'Supposing', 'Given that', 'Contingent on', 'In the occurrence of', 'Hinging on'. There are many more possibilities and slight variations on these patterns, some more familiar than others. We can include all of them in our initial pattern set and filter out those that are not prominent in real contracts as we investigate RQ3.

There are a handful of synonyms in this list that require a special discussion. Words like 'upon' and 'once' were also noted as synonyms for 'after', and were part of our discussion on the `HappensAfter` refinements. This presents another interesting ambiguity challenge. The words 'if' and 'after' are different enough in meaning *in general*, but in certain contexts, they have common synonyms ('upon', 'once'). Given the NL statement 'Seller must deliver the goods *upon* payment by the buyer', do we interpret the 'upon' phrase as indicating a conditional operation or a 'happens after' operation? Similar arguments could be made for 'once' and even the word 'after' itself. We can attempt to resolve this ambiguity by acknowledging that for our purposes, these statements will mainly occur in the context of *norms*. Consider a non-normative statement of our previous example: 'the seller delivered the goods upon payment by the buyer'. If we were to map this to SYMBOLEO-like predicates, the `HappensAfter` arguably makes a little more sense. However, if we consider the event in the context of an obligation, as in our original *normative* statement, the conditional interpretation makes more sense. For now we will consider both interpretations as possibilities, and root our decision on which mapping is more appropriate in the investigation of real contracts in RQ3.

Refinements with Additional Conjuncts

Much of the technical linguistic terminology we have been using can be confusing, so we take a moment to clarify two potentially confusing terms, which sound similar: Conditional Adjuncts and Additional Conjuncts. A *conditional adjunct* is a *linguistic* term referring to an adjunct that begins with a conditional preposition, such as 'if', 'when', or 'in case of'. An *additional conjunct* is a term that refers to adding a new predicate to an existing SYMBOLEO proposition. For example, consider the obligation: `O(partyA, partyB, true, Happens(eventX))`. The consequent is a proposition consisting of a single `Happens` predicate. If we were to add another predicate to this proposition using the AND operator (conjunction), it would be considered an additional conjunct: `O(partyA, partyB, true, Happens(eventX) AND Happens(eventY))`.

This illustrates the second scenario of adding a predicate conjunct: when we already have a predicate in a proposition and want to add *another*. This SYMBOLEO operation can trivially correspond to the conjunction 'and' followed by the statement of an event ('and' EVENT). For example, our original statement might be: 'The seller must deliver the goods to the buyer', with the SYMBOLEO `O(seller, buyer, true, Happens(deliver_goods))`, and we might refine it by adding another event: 'The seller must deliver the goods to the buyer *and provide a receipt to the buyer*'. The corresponding SYMBOLEO is: `O(seller, buyer, true, Happens(evt_deliver_goods) AND Happens(evt_provide_receipt))`. Other synonyms for 'and' that can also express conjunction include 'along with', 'in addition to', and 'as well as'.

The notion of a conjunction gets complicated when we consider how language can be used to *hide* much of the information in the expression of events in conjunction. Consider the sentence: 'you must wash and dry the dishes, sweep the floor, or feed the cat and dog.' If we were to break this statement down into a atomic standard events, it might look something like this: (wash_dishes AND dry_dishes) OR (sweep_floor) OR (feed_cat AND feed_dog). Additional events can be suggested by adding the conjunct between verbs (wash and dry), between nouns (cat and dog), or by the use of commas. Our standard event specification is not equipped for these complexities, so we will exclude operations arising from conjunctions from our analysis. Future work may involve building a more complex event specification, at which point it may be possible and desirable to support these operations.

4.3.6 Domain Model Updates

SYMBOLEO contracts contain a domain model and a contract specification. The domain model contains the *building blocks* for the contract specification, which contains the norms. The events, assets, and roles that make up the norms in the contract specification must all first be defined in the domain model and then instantiated as declarations. The reason we bring this up is that in the application that we are attempting to create, it will be changes to the contract specification that drive any changes to the domain model. The user will be focused on refining text corresponding to the *norms*, and they would not be defining a new event, asset, or role *unless it is within the context* of refining a norm. We will now explore the general semantics behind these operations related to the domain model.

Adding Domain Model Events

In the cases where the refinement involves the statement of an event (e.g., refining to `WhappensBeforeEvent` or adding an event-based conditional), there is an added consideration for the SYMBOLEO update. If the user enters a brand new event (i.e., an event that is *not* in the domain model), then we must first add this event to the domain model. Consider the example 'partyA must approve the transaction [PARAMETER]', and suppose the PARAMETER is filled with `'before partyB provides an inspection report'`. Since the approval event is in the template T, we can assume it will be in the domain model in S(T). The event in the refinement of providing an inspection report, however, may *not* be in the domain model, as it could be a completely new event introduced by the user. The corresponding refined predicate may be something like `WhappensBeforeEvent(evt_approve_transaction, evt_provide_report)`, but the declaration evt_provide_report may not yet exist. Therefore, we must first add this event to the domain model, and then create a corresponding declaration in the contract specification. Only then will we be able to properly refine the norm into a predicate. In our example, the added domain model event that would be created might be as follows:

```
ProvideReport isAn Event with
   agent: Role, recipient: Role, report: InspectionReport;
```

The corresponding declaration might then be:

```
evt_provide_report: ProvideReport with
   agent := partyB, recipient := partyA, report := inspection_report;
```

Note that this new event *also* introduces a new asset type, the InspectionReport. This would also need to be added to the domain model and as declaration, as follows:

- **Domain Model:** `InspectionReport isAn Asset;`
- **Declaration:** `inspection_report: InspectionReport;`

The asset declaration can now be used in the new event declaration, which can then be used in the predicate refinement. Any time a new event is introduced in the process of refining a norm, we will need to perform these operations in addition to the direct norm refinement. The operations of adding new domain classes and their corresponding declarations therefore require *no new NL patterns*. We need only note that certain refinements that we have already introduced will result in these additional SYMBOLEO operations, namely those that include an EVENT.

In the case where our refinement deals with dates (e.g., before DATE), we have noted previously that it would require the addition of a Contract Specification Parameter (CSP) to the contract signature, which holds the specified date value. There may be an argument in favour of *also* adding a date-based declaration property to an existing event declaration for such a refinement, but that would require updating *all* instances of the Domain Event with the new property.

Adding Dynamic Properties

We've covered the addition of a completely new event, but what if the user wants to simply refine an *existing* event in a non-temporal way? For example, they may want to specify that an event must be done in a certain manner, with a certain instrument, or at a certain location. These types of refinements can be captured by the various adjuncts that we've discussed, as shown in the following examples (adjuncts are italicized):

- **Instrumental PP:** The buyer must pay the seller $100 *using a credit card*
- **Locative PP:** The goods must be delivered *to the buyer's address*
- **Manner Adverb**: The package must be handled *carefully*

SYMBOLEO is primarily concerned with the flow of events, which is why it includes very specific conditional and temporal constructs. It is conceivable that these other types of refinements (instrumental, locative, etc.) might be useful in a smart contract scenario. For example, it may be important for our smart contract to know that a payment was made with a credit card. However, we consider these non-temporal/non-conditional details to be of secondary importance, and we will therefore exclude these types of refinements from further consideration. We acknowledge that these may be introduced in future work, and we explore ideas for these operations in Appendix A.

4.3.7 Summary of Operations

It may appear counter-intuitive at first that refining a NL statement using different types of adjuncts can result in vastly different operations on the SYMBOLEO contract. Consider again the simple sentence: 'The seller must deliver the goods to the buyer [PARAMETER]'. Now consider three ways of refining it:

- **Temporal adjunct:** The seller must deliver the goods to the buyer *before April 15, 2023.*
- **Conditional adjunct:** The seller must deliver the goods to the buyer *if payment has been completed.*
- **Locative adjunct:** The seller must deliver the goods to the buyer *at 123 main street.*

The temporal adjunct results in a predicate refinement. The conditional adjunct results in an added antecedent, and – since an event is specified – potentially a new domain model event and a new declaration. The locative adjunct results in updating the delivery event on the domain model. This difference in outcome is largely due the fact that SYMBOLEO is event-based, and therefore has special emphasis on temporal and conditional refinements. Here it is worth re-iterating that ultimately a contract author does not see or care about these operational details. All they care about is the CNL

interface. They would choose a parameter to refine, and step through a guided process of entering one of these patterns that we are in the process of defining. They only see the NL contract, but under the hood, the appropriate Symboleo would be generated.

We summarize our results of RQ2 by presenting the working CNL resulting from our linguistic analysis in this section. The CNL consists of two components:

1. A set of mappings between a NL pattern class and a Symboleo operation for the selected list of specific Symboleo operations, shown in Table 4.1. Each pairing is accompanied by a sample NL refinement.

2. A dynamic EBNF grammar containing the rules for *specifying* the pattern class components, shown in Listing 4.4.

We emphasize that this working CNL represents a set of *heuristics* based on careful linguistic analysis. The complexity of NL resists any strict *rules* about representation, so we are confined to these heuristics. While this working CNL *could* be useful for our system, its construction is not rooted in *real* contract data, so its value may be limited. This is precisely what is addressed in RQ3.

Pattern Class	**Symboleo Operation**	**Sample Refinement**
P_BEFORE_S DATE	Happens → ShappensBefore	before March 30, 2024
P_BEFORE_PF POINT_FUNCTION	Happens → WhappensBefore	prior to 2 weeks after the payment is completed
within TIMESPAN P_AFTER_W EVENT	Happens → WhappensBefore	within 2 weeks of the contract terminating
P_BEFORE_E EVENT	Happens → WhappensBeforeEvent	before payment is completed
P_AFTER_E EVENT	Happens → WhappensBeforeEvent	after the contract terminates
P_AFTER DATE	Happens → HappensAfter	after March 30, 2024
P_AFTER_PF POINT_FUNCTION	Happens → HappensAfter	after 2 weeks following shipment of goods
INTERVAL	Happens → HappensWithin	for 1 year following contract termination
CONDITIONAL_T EVENT	Add Trigger	When the document is disclosed
CONDITIONAL_A EVENT	Add Antecedent	In the event that the buyer cannot complete payment

Table 4.1: Initial CNL mappings

Listing 4.4: Initial working grammar

```
<EVENT> ::= <SUBJ> <VP>
<SUBJ> ::= simple_np
<VP> ::= [<ADVB>] (<TVP> | <LVP> | <IVP>) [<PP>]{1-2}
<ADVB ::= adverb
<TVP> ::= transitive_verb <DOBJ>
<DOBJ> ::= simple_np
<LVP> ::= linking_verb <PRED>
<PRED> ::= adjective
<IVP> ::= intransitive_verb
<PP> ::= preposition <POBJ>
```

```
<POBJ> ::= simple_np

<TIMEPOINT> ::= <DATE> | <EVENT> | <POINT_FUNCTION>
<DATE> ::= (various date formats)
<POINT_FUNCTION> ::= <TIMESPAN> <P_PF> <EVENT>
<TIMESPAN> ::= <TIME_VALUE> <TIME_UNIT>
<TIME_VALUE> ::= Number
<TIME_UNIT> ::= "hours" | "days" | "weeks" | ...
<P_PF> ::= "before" | "prior to" | "after" | ...

<INTERVAL> ::=
    "between" <TIMEPOINT> "and" <TIMEPOINT>  |
    ["from" <TIMEPOINT>] "until" <TIMEPOINT> |
    "during" <TIME_PERIOD> |
    "for" <TIMESPAN> P_AFTER_I <TIMEPOINT>
<P_AFTER_I> ::= "following" | "after" | ...
<TIME_PERIOD> ::= (various specifications)

<P_BEFORE_S> ::= "before" | "by" | "prior to" | ...
<P_BEFORE_PF> ::= "before" | "earlier than" | "prior to" | ...
<P_AFTER_W> ::= "after" | "of" | ...
<P_BEFORE_E> ::= "before" | "prior to" | ...
<P_AFTER> ::= "after" | "following" | "later than" | ...
<P_AFTER_PF> ::= "after" | "following" | "later than" | ...
<P_AFTER_E> ::= "after" | "following" | "once" | "upon" | ...

<CONDITIONAL_A> ::= "if" | "in the event [that]" | "in case" | "
   once" | "upon" | ...
<CONDITIONAL_T> ::= "when" | "whenever" | ...
```

4.4 RQ3: An Evidence-based CNL

4.4.1 Outline of RQ3

We now have a working CNL that can be used for refining a NL contract template. This CNL consists of a dynamic EBNF grammar and a set of mappings between patterns based on this grammar and SYMBOLEO operations. This CNL was constructed using a *general* semantic treatment, and thus it can represent a useful set of heuristics for identifying SYMBOLEO operations from refined norms. We now want to refine this CNL to align more specifically with semantics that appear in *real* NL contracts. For example, we have established that a refinement like '`before March 30, 2024`' likely corresponds to the SYMBOLEO operation that maps a `Happens` predicate to a `ShappensBefore` predicate. We want to see if this textual pattern (before DATE) is present in real contracts *and* verify that it corresponds to this target SYMBOLEO operation. Our strategy will be to take a set of NL refinements for norms found in real contracts, and for each refinement, identify the general pattern it expresses, as well as the SYMBOLEO operation to which it corresponds.

This dataset of manually-created mappings will be used to construct and evaluate our CNL.

We cannot simply look for the presence of a particular NL pattern in a real contract and *assume* that it corresponds to a certain operation. There may be false positives where some pattern keywords are present in the NL contract, but do *not* correspond to the assumed operation. Consider the word 'if'. We have identified it as a conditional keyword, but it does not always have a conditional role, as seen in the example 'Find out if the guests have all arrived'. This is why we need to consider both the NL expression *as well as* the corresponding SYMBOLEO operation. The analysis will proceed as follows:

1. We identify a relevant and manageable dataset of real contracts.
2. From this dataset, we extract sentences that contain potential norms (obligations and powers).
3. From this set of norms, we identify those that contain relevant refinements. Based on our heuristics from RQ2, a *relevant* refinement is one that takes the form of conditional or temporal adjunct.
4. For each refinement, we identify the operation in SYMBOLEO, identified in RQ1, to which it corresponds.
5. Once we have performed this process on all refinements, we will identify the most common patterns and mappings, which will form our CNL.

This analysis will allow us to refine our working CNL from RQ2 into something more specific by *excluding* patterns that do not appear in real contracts. Conversely, it may also *expand* our grammar if we discover refinements on obligations that were not covered in our semantic analysis in RQ2. This analysis represents work towards a useful dataset as well as a *methodology* for generating such a dataset, which can be useful for other applications in the SYMBOLEO ecosystem, or for iterating on the CNL as requirements change.

4.4.2 Building the Dataset

As we walk through the CNL construction process, we will be referencing various artifacts produced at each stage of the construction. Illustrative examples will be provided where necessary, but further technical details and code samples of this procedure can be found in Appendix B.

Initial Dataset

The Electronic Data Gathering, Analysis, and Retrieval (EDGAR) dataset [2], contains contracts from publicly-traded companies that are required to report filings to the United States Securities and Exchange Commission (Securities and Exchange Commission (SEC)).

[2] `https://www.sec.gov/edgar/search-and-access`

These contracts are free and open to the public, and they have been used for a variety of related work in the legal NLP domain, including the development of the Contract Understanding Atticus Dataset (CUAD) dataset [43]. This dataset consists of 510 contracts from EDGAR that were carefully analyzed and annotated by legal experts with useful legal information. While we do not require these annotated labels, this dataset is also made easily available for public access, and the contracts in this dataset come from a wide variety of industries (e.g., banking, transportation, employment, etc.). For these reasons, it is useful for our purposes.

The focus of Symboleo is on monitoring the execution of contracts, so there is particular interest in norms that may be amenable to such automation, such as those related to payments and scheduling. Since this is fairly novel work, we are interested in a *broader* exploration of norms found in contracts. We therefore do not restrict our dataset to any specific industry, although this may be a feature of future iterations. Furthermore, this phase of research potentially involves careful manual analysis of entire contracts in order to understand the context of the norms contained within. We therefore limit ourselves to shorter contracts (between 500-3000 words), which reduces our dataset to a more manageable 109 contracts.

Extracting Norms

From this filtered dataset, we attempt to extract any sentences that contain norms. This is done heuristically by extracting all sentences that contain a *trigger word* for an obligation, a technique borrowed from much of the related work discussed earlier. Trigger words for norms include (but are not limited to) 'must', 'shall', 'may', and 'required' (be careful not to confuse this usage of the word 'trigger' with the Symboleo concept of a trigger). Filtering the sentences in the 109 contracts to those that only contain these trigger words results in a set of over 2700 sentences.

We are only interested in norms that contain refinements that are relevant to Symboleo (e.g., temporal and conditional refinements, as discussed in RQ2). We can facilitate the identification of norms that contain these types of refinements by performing further keyword searches on these sentences. For example, we could tag any mentions of the keyword 'before' in these sentences, which may help us more easily identify certain types of refinements. One possibility would be to *remove* norms that contain *none* of these heuristic keywords. For example, suppose our keyword list contained only the words 'until', 'before', 'after', and 'within'. Suppose then that we automatically filtered out any sentences that did not contain *any* of these four words. While this would make analysis less tedious, we note that this would cause us to miss out on *false negatives*, i.e., norms that contain refinements that make use of patterns that we *missed* in our general semantic analysis. Therefore, we include all of sentences, and simply *tag* the sentences that contain these keywords. This means that we must still manually analyze each potential norm, but the tags may make this analysis easier.

Identifying Refinements and Operations

We now have a list of about 2700 sentences that are marked with our heuristics from RQ2. We must now manually filter this list down to sentences that meet the following properties:

- **The sentence is an actual norm:** This involves filtering out sentences that were captured as false positives by our norm trigger words. For example the sentence 'The agreement begins on may 1st, 2015' would be an example of a false positive, since it mis-identified the word 'may' as corresponding to a norm.
- **The norm contains a temporal or conditional refinement:** This is where we can make use of our heuristic keywords from RQ2. For example the sentence 'Domini shall pay the entire salaries and wages of all of the trust's trustees and officers' represents an obligation, but there is no refinement. The sentence 'upon termination of this agreement, Emdeon shall remove the content from the software' expresses a norm *and* the norm includes the refinement 'upon termination of this agreement'. The refinement must furthermore take the form of an adjunct in that it is *optional*, as it does in this example. Norms that do not contain an adjunct refinement will be filtered out.
- **The norm refinement corresponds to a Symboleo operation**: Once we have identified the norms with refinements, we want to know which specific SYMBOLEO operation, if any, the refinement corresponds to. Once again, this is where we can use our heuristics as a *guide*, but the final judgment should be done manually, and ideally with expert input. We may come across refinements that correspond to operations that SYMBOLEO currently does not support (such as those related to frequency), and these will be omitted from our analysis.

Completing the Dataset

The result of this manual verification process is a set of 294 sentences that meet the above properties. For each sentence, we have verified that it corresponds to a norm, extracted the refinement, as well as the SYMBOLEO operation to which the refinement corresponds. We will discuss limitations to this process later on, but for now we will note that input from both SYMBOLEO and legal experts would render these results more reliable. To this end we also add norms from a small set of realistic fully-specified contracts, some of which *have* received some input from experts, including our meat sale contract example from Listing 2.1. There are many online services that provide traditional FITB contract templates for a variety of domains. We choose a property management contract, a rental contract, and an independent contractor contract obtained from Signwell [3], and add the relevant norms/refinements contained within to our dataset. This inclusion adds a degree of authority to our dataset. In total, this gives us a dataset of size 320. We list a few examples of refined norms (refinement in italics) with the corresponding general SYMBOLEO operation:

[3] `https://www.signwell.com/`

- Party B shall pay the license fee to the account designated by Party A *before December 31.*
  - **Operation**: `Happens` → `ShappensBefore`
- *In the event an audit discloses non-compliance with this agreement*, customer shall pay to Cisco the appropriate license fees.
  - **Operation**: Add an antecedent
- *During the period beginning October 1, 2009 and ending March 31, 2010*, charity tunes shall not enable another program sponsorship...
  - **Operation**: `Happens` → `HappensWithin`

4.4.3 CNL Construction Example

We are now ready to construct our CNL using this dataset. An important metric that we can associate with our CNL is *pattern coverage*; Given a norm refinement from an arbitrary NL contract, can this refinement be expressed in our CNL grammar, *and* does it map to our predicted Symboleo operation? In order to quantify this metric, we therefore randomly split our dataset into a construction set (80%) and a test set (20%). The test set will not be considered for the construction of the CNL, but will be used for evaluating the coverage once it is complete.

To construct the CNL from our construction set, we first extract the *pattern* from each of our refinements. As discussed in the previous section, a *pattern* involves the replacement of our temporal building blocks (timespans, events, time periods, dates, etc.) with rule names. The pattern of the refinements in our examples (previous section) are before DATE, in the event EVENT, and during TIME_PERIOD, respectively. Once these patterns are identified, we will look for the broader *pattern classes*, which can include one or more patterns that are synonymous with each other. The patterns and pattern classes introduced in RQ2 provide a starting point for this identification, but the careful manual analysis of the real contracts will ultimately result in refining these classes for our final CNL.

For example, if we find that the patterns before DATE and prior to DATE often result in the same operation (`Happens` → `ShappensBefore`) in our construction set, and the semantic meanings are very similar, then we can group these patterns under the *pattern class* of P_BEFORE_S DATE, which aligns with our semantic analysis in RQ2. We can then count the number of examples from our construction set where a P_BEFORE_S DATE refinement maps to this operation. A high count provides evidence that this mapping should be included in our CNL. There may be a possibility that we find an example of a refinement matching the pattern class P_BEFORE_S DATE, but which maps to a *different* Symboleo operation. While this seems unlikely from a semantic analysis, we cannot rule it out, since we do not want to make unnecessary assumptions about the regularity of NL. If we found such examples, then we could include these additional mappings in our CNL,

and we would need to look more closely at the context to determine *which* SYMBOLEO operation the matching refinement will map to.

Also possible is the case that we find text that matches the pattern class, but does *not* correspond to *any* refinement. These cases do not concern us for now, but may need to be addressed in future work that attempts to further automate the identification of valid norm refinements. Below are some examples of refined norms that result in the operation `Happens` → `ShappensBefore`. The pattern expressed by the refinement is in bold:

- **by DATE**: Charity tunes shall use its best efforts to update its music catalogue with available mp3 files *by June 30, 2009*.
- **before DATE**: If ten (10) music venues sign a contract with VNUE *before January 16, 2016*, promoter will receive an additional bonus of three hundred thousand (300,000) shares of vnue common stock.
- **until DATE**: Shi farms will use its best efforts to quarantine product *until delivery date* as agreed by the parties.

We once again confirm that these are all *optional* refinements. The patterns can be found by simply replacing the specific date (or date parameter) with the word DATE. Next, we want to consider the more general *pattern classes*. For the first two examples, the patterns are synonymous, and therefore belong to the same pattern class P_BEFORE_S DATE, where the rule P_BEFORE_S, will include 'before' and 'by'. We find 14 such examples in our construction set of general patterns that have synonymous meaning and are therefore also captured by this pattern class. These examples show the value of performing our semantic analysis in RQ2, as it provides a foundation for our patterns and pattern classes.

The third example, using 'until', has a slightly different meaning. As we have previously discussed, 'until' suggests a situation or duration, and will therefore need to be framed as a negated event in SYMBOLEO (e.g., `not Happens(evt_break_quarantine)`). We had previously included this in our `HappensWithin` refinement operation, but in this case it corresponds to a `ShappensBefore` refinement. This example highlights a weakness in our semantic analysis of RQ2, and shows the importance of performing the analysis on real contracts instead of relying purely on heuristics. Since this pattern is *not* synonymous with P_BEFORE_S DATE, we will keep it as a separate pattern class in our CNL. We therefore include the following mappings in our CNL:

- P_BEFORE_S DATE: `Happens` → `ShappensBefore`
- until DATE: `Happens` → `ShappensBefore`

We provide the SYMBOLEO specification for our first example as an illustration. Note that the date value of 'June 30, 2009' would be passed in as an argument with the name `ob_update_date` into the contract instantiation:

- **Initial Symboleo**: `O(Charity tunes, customer, true, Happens(evt_update_catalogue))`

- **Updated Symboleo**: `O(charity tunes, customer, true, ShappensBefore(evt_update_catalogue, ob_update_date))`

We will apply this same process to the rest of our operations to construct our full CNL. For each operation, we will provide illustrative examples from the dataset, discuss the main patterns and the pattern classes, and state the final additions to our CNL. We note that this treatment will only address the operations that involve a direct norm refinement. This excludes the extra SYMBOLEO operations of adding any required domain classes or declarations. These extra operations are going to be present whenever an event or asset is specified that *does not already belong to the domain model*, and they will become important during the discussion of the SYMBOLEONLP implementation. For this discussion, however, we are only interested in the direct refinement on SYMBOLEO obligation and power specifications. Before we address the operations, we make some important notes on the complexity of event specification as it relates to this discussion.

4.4.4 Event Complexity

We previously defined the notion of a *standard event*, and noted that this is the format that will be supported by SYMBOLEONLP. In this section we will see many cases of events that are far more complex than this standard event format. Consider the following events (italicized) found among this dataset:

1. Dolphin agrees to complete its photo-editing services within 14 days of *receiving the original digital photo files.*
2. BOSCH reserves the right to share rights given unless *it disrupts and/or interferes with CLIENTS business and/or productivity.*
3. If *you believe that the Company has billed you incorrectly*, you must contact Company...
4. Once *the common stock has been registered, or, after the one year period applicable under Rule 144, whichever occurs first*, the Company...

The first example has an implied subject (Dolphin) for the recipient of the files. If we were to specify this as a standard event, it would be: 'Dolphin receives the original digital photo files'. 'Receives' is a transitive verb with 'original digital photo files' acting as the direct object. Adding the explicit subject is useful in that it replaces implied information (thus reducing ambiguity), though this may be considered redundant in regular speech.

The second example highlights two more challenges. First of all, the event begins with the pronoun 'it'. This pronoun is often used as a co-reference to refer to something previously mentioned in the text. In this case, 'it' may refer to the company itself (BOSCH) or the act of sharing rights. Just as we don't want to allow implied information, we also want to eliminate these co-references, which have potential for introducing ambiguity. We note that there are NLP tools and techniques that can resolve these co-references [4],

[4] https://pypi.org/project/coreferee/

meaning the word ‘it’ could potentially be made acceptable within our CNL. The second challenge with this sentence is the use of ‘and/or’ – in fact it is used twice in this event. This is therefore hiding at least four different standard events, which we cannot represent in our current formulation of a standard event. We can still use these events as test cases for SymboleoNLP, but we note the events will be simplified to standard form, resulting in a loss of information. For this particular example, we might reduce it to ‘unless BOSCH disrupts CLIENTS productivity’, which can still be used to test the unless EVENT pattern, which will be introduced later on.

The third example uses a subordinate clause (‘the Company has billed you incorrectly’) within a main clause (‘you believe’) linked by the subordinator ‘that’. One potential way of capturing a subordinating conjunction might be to have an optional subordinate clause property within an event. This property would take the form of another event, so we would have an event within an event. The final example we show includes a disjunction (or) with the additional qualification of ‘whichever comes first’. This is not unusual to see in contracts, but incorporating support for this type of statement might require new constructs in Symboleo itself, perhaps a new predicate type to handle this scenario.

These examples highlight just a few of the many complexities involved with the specification of events in NL that we will not be handling. This introduces a limitation on our work, but examples like these will motivate future work on this research. For example, the event grammar and specification may be expanded to include concepts such as co-references and subordinate clauses, which would allow more events to be represented in the system. For this thesis, we are focusing on the more general patterns of refinement. As we proceed through this section, complexities associated with these events may suggest different interpretations of how it might be represented in Symboleo. However, we will be treating most events in a simplified manner: Whenever we come across an event within a refinement, we can imagine replacing it with the *dummy event* of ‘perform event X’. While we could simply apply this replacement on all events mentioned in this section, we leave in the intact NL events to highlight the diversity of event expression in NL contracts. Some are slightly simplified to avoid unnecessary wordiness. With this dummy substitution, our previous examples would become:

1. Dolphin agrees to complete its photo-editing services within 14 days of *Dolphin performing event X*
2. BOSCH reserves the right to share rights given unless *BOSCH performs event X*.
3. If *you perform event X*, you must contact Company...
4. Once *Company performs event X*, the Company...

We will re-visit the specification of events from an implementation perspective when we describe the SymboleoNLP application. We now move on to the analysis of the Symboleo operations with respect to the construction of the CNL.

4.4.5 Predicate Refinements

This section covers the pattern classes related to the four categories of predicate refinements selected earlier.

Happens → WhappensBefore

Examples of norms, their refinements (in italics), and their patterns (in bold) for this operation include:

- **within TIMESPAN of EVENT**: Dolphin agrees to complete its photo-editing services *within 14 days of receiving the original digital photo files.*
- **TIMESPAN following EVENT**: Payment to Porex of undisputed fees shall be due *X days following Cerus' receipt of the invoice submitted by Porex.*
- **at least TIMESPAN prior to EVENT**: Display materials from the sponsor must be delivered to the Tian-He stadium *at least two days prior to the event.*

Our dataset includes many refinements that correspond to this operation, and many of these can be grouped into a relatively small number of pattern classes. Some common general patterns include within TIMESPAN of EVENT, within TIMESPAN after EVENT, within TIMESPAN following EVENT, all of which align with the pattern class within TIMESPAN P_AFTER_W EVENT identified in RQ2. In some examples, an anchor event is *implied*, such as: 'In the event of delay by vehicle malfunction, party b shall settle such malfunction or accidents *within half an hour*'. The implied event here is the delay (*within half an hour of the delay*). We include examples like this in our within TIMESPAN P_AFTER_W EVENT pattern class as well.

Another pattern class identified in RQ2 includes P_BEFORE_PF POINT_FUNCTION, where a POINT_FUNCTION can specify a relative time *before or after* another point in time. We found many cases of such point functions, but few adhered to our initial pattern. The much more common case simply used a POINT_FUNCTION, often with 'at least' appearing at the start (e.g., 'The company must complete the services at least 2 weeks prior to contract termination'). We also must differentiate between the cases where the relative point function is *before or after* the anchor event. In the previous example, the 3 weeks timespan is *before* the anchor point (contract termination). In the norm 'Payment shall be made to the appropriate party *30 days from the end of the calendar month*', the 30 day timespan is *after* the anchor point (end of the calendar month). Note, however, that they both still correspond to the obligation being required to happen *before* the specified timepoint. We therefore split our initial pattern class from RQ2 into two separate mappings. Both will map to the `Happens` → `WhappensBefore` operation, but since they are semantically distinct, they must be separate pattern classes. Our mappings from this section therefore include:

- within TIMESPAN P_AFTER_W EVENT: `Happens` → `WhappensBefore`

- at least TIMESPAN P_BEFORE_T EVENT: Happens → WhappensBefore
- at least TIMESPAN P_AFTER_T EVENT: Happens → WhappensBefore

We keep the P_AFTER_W rule from RQ2, but replace the P_BEFORE_PF rule with the P_BEFORE_T and P_AFTER_T rules. These are specified as follows, based on their presence in our construction set:

- **P_AFTER_W**: after, following, from, of
- **P_BEFORE_T**: before, prior to
- **P_AFTER_T**: after, following, from

As a concrete illustration, consider the Symboleo specification of our first example, which uses the within TIMESPAN P_AFTER_W EVENT pattern class. Recall that we are assuming the Symboleo contract contains declarations and domain classes corresponding to the required roles, assets, and events:

- **Initial Symboleo**: `O(Dolphin, client, true, Happens(evt_complete_services))`
- **Updated Symboleo**: `O(Dolphin, client, true, WhappensBefore(evt_complete_services, Date.add(evt_receive_files, 14, days)))`

Happens → WhappensBeforeEvent

Examples of refined norms that correspond to this operation include:

- **before EVENT**: Owner shall, *before commencement of each voyage by any vessel*, exercise commercially reasonable efforts to ensure that such vessel is seaworthy and in good operating condition.
- **prior to EVENT**: Distributor shall obtain an RMA number *prior to returning any product to Cisco.*
- **until EVENT**: Prime agrees to fund additional shareholder loans equal to the amount repaid by the shareholder *until each component of project completion is satisfied.*

These examples exhaustively highlight all three patterns observed in the construction set for this operation. The before EVENT and prior to EVENT can be joined into the common pattern class P_BEFORE_E EVENT, where P_BEFORE_E includes 'before' and 'prior to'. Similar to what we saw with ShappensBefore, the 'until' keyword results in a semantically distinct pattern from these other cases, and we will therefore consider it to be part of its own pattern class.

We also previously noted the possible case where we can have an after EVENT pattern, which results in a WhappensBeforeEvent refinement. We found one such example in

our dataset: 'If dividends are credited *after premiums can no longer be paid under this contract*, dividends will be paid in cash'. In this case, we would simply need to swap the argument order on the `WhappensBeforeEvent` predicate. While there is only a single example of this, it is semantically distinct, so we must include it in our CNL. We note, however, that this particular refinement is on the *conditional* statement, rather than the *consequent*. We will return to this fact when we discuss the conditional refinements. The mappings for this operation include:

- P_BEFORE_E EVENT: `Happens` → `WhappensBeforeEvent`
- until EVENT: `Happens` → `WhappensBeforeEvent`
- P_AFTER_E EVENT: `Happens` → `WhappensBeforeEvent` (with arguments swapped; Only applies for refinement of *conditional* statements)

We provide the Symboleo specification for our first example:

- **Initial Symboleo**: `O(Owner, company, true, Happens(evt_ensure_seaworthy))`
- **Updated Symboleo**: `O(Owner, company, true, WhappensBeforeEvent(evt_ensure_seaworthy, evt_commence_voyage))`

Happens → HappensAfter

We found only two examples of refined norms that correspond to this operation in our construction set:

- **Following TIMESPAN from EVENT**: *Following the expiration of 90 days from the termination or expiration of this agreement*, the company shall cease usage of all advertising materials which contain the professional's image.
- **later than DATE**: In no case may any PIN code be redeemed later than March 31, 2010.

The first example could be included in our P_AFTER_PF POINT_FUNCTION pattern class from RQ2, but once again we want to distinguish cases where the timespan is either *before or after* the event. We will therefore once again split up this pattern class into two separate ones, and re-use our P_AFTER_T and P_BEFORE_T rules. The second example highlights an obligation *not* to perform an event after a certain date. The pattern can be included in the P_AFTER DATE pattern class. Our pattern classes therefore include:

- P_AFTER DATE: `Happens` → `HappensAfter`
- P_AFTER_PF TIMESPAN P_AFTER_T EVENT: `Happens` → `HappensAfter`
- P_AFTER_PF TIMESPAN P_BEFORE_T EVENT: `Happens` → `HappensAfter`

We did not actually observe cases of this final pattern class in our dataset. However, it logically follows from other patterns discovered, so we include it. We provide the SYMBOLEO specification for our first example, which uses the P_AFTER_PF TIMESPAN P_AFTER_T EVENT pattern class:

- **Initial Symboleo**: `O(company, professional, true, Happens(evt_cease_advertising))`
- **Updated Symboleo**: `O(company, professional, true, HappensAfter(evt_cease_advertising, Date.add(Terminated(self), 90, days))`

Happens → HappensWithin

Examples of refined norms that correspond to this operation include:

- **during TIME_PERIOD**: *During the term of this agreement*, licensee shall use the licensed mark only to the extent permitted under this license territory.
- **for TIMESPAN following EVENT**: *For 24 months following termination of this agreement*, licensee shall specify on all public-facing materials that licensee is no longer operating under the licensed mark.
- **during period beginning TIMEPOINT and ending TIMEPOINT**: *During the period beginning October 1, 2009 and ending March 31, 2010*, charity tunes shall not enable another program sponsorship...

Many of the patterns that map to this operation are already captured in our specification for an INTERVAL in Listing 4.3, but we will parse out the components in more detail. Our first example highlights a very common duration expression of 'during the term of the agreement', or some variation thereof. This refinement could arguably be considered superfluous, as it may be implied that anything stipulated in the contract persists *precisely* throughout the period of agreement and neither before nor after. Nevertheless, its ubiquity suggests that we include the pattern in our final CNL. This is captured by the during TIME_PERIOD pattern, which can be generalized to the pattern class P_DURING TIME_PERIOD. P_DURING includes 'during', 'within', and 'throughout', as these are other examples that appear in our contracts.

Many of the patterns that map to this operation can be captured in the general pattern class for TIMESPAN P_AFTER_I EVENT. This denotes that some situation holds for a certain timespan following the occurrence of an anchor event. We also found a few cases where a norm is refined using *both* patterns (e.g., 'during the term of the agreement *and* for 90 days following the termination of the agreement'). We consider these to be two separate refinements, but we currently do not consider the possibility of refining a norm component with more than one refinement, so this scenario can be a motivator for future work. We also note that examples like this suggest a *surviving obligation*, since the time frame extends *beyond* the agreement. Our third example delineates an interval of time using two explicit dates. This could be synonymously captured by the simpler pattern

class of between TIMEPOINT and TIMEPOINT, which includes the assumption that the first TIMEPOINT is before the second TIMEPOINT.

We noted that all of these cases could be captured in the pattern class of INTERVAL, but since each interval specification has a different semantic meaning, we will capture each one as its own pattern class. The pattern classes for this operation include:

- 'between' DATE 'and' DATE: `Happens` → `HappensWithin`
- 'for' TIMESPAN P_AFTER_I EVENT: `Happens` → `HappensWithin`
- P_DURING TIME_PERIOD: `Happens` → `HappensWithin`

Note that we are also able to get rid of both the POINT_FUNCTION and TIMEPOINT specifications in lieu of more concrete rules. We went through the process of more precisely breaking down the various other usages of a point function through rules like P_BEFORE_T and P_AFTER_T, and thus we no longer require the vaguer point function. We therefore reduce the idea of a TIMEPOINT to be either a DATE or an EVENT. We provide the SYMBOLEO specification for our second example. Note that this would need to become a *surviving obligation*, since it persists beyond the contract termination:

- **Initial Symboleo**: `O(licensee, licensor, true, not Happens(evt_cease_specifying_mark))`
- **Updated Symboleo**: `O(licensee, licensor, true, not HappensWithin(evt_cease_specifying_mark, Interval(Terminates(self), Date.add(Terminates(self), 24, months))))`

4.4.6 Adding Predicates

Adding a Trigger

We have suggested that we use conditional keywords such as 'when' and 'whenever' to correspond to triggers, since these suggest the notion of recurrence. This is merely a heuristic, so we would expect there to be both false positives examples and false negative examples of this. A false positive would be a use case of 'when' that does *not* suggest a recurrence (e.g., 'The Client will be invoiced *when the services are complete*', assuming the services are only completed *once*.). A false negative would be a use case of *other* conditionals that *do* suggest a recurrence (e.g., '*in the event of delay by vehicle malfunction* party b shall settle such malfunction...', assuming that there could be *multiple* delays). This distinction can be dependent on the context of the specific event being discussed, which we have largely abstracted away. We will therefore re-emphasize this as a limitation and continue using our heuristic. Other examples of the when EVENT pattern include:

- WebMD shall promptly notify Emdeon *when a new update is available.*
- *When the agreement ends*, each party shall return all copies of any such information to the other...

There were many cases for both 'when' and 'whenever' that seem to suggest a conditional, but are too vague to be considered useful in terms of smart contracts, such as 'whenever possible' or 'when necessary'. This is the type of language that was cautioned against in the development of a legal CNL [26] in our related work. We disregard these cases, and limit the CONDITIONAL_T to simply include the keyword 'when'. We retain our pattern for the operation of adding a trigger as CONDITIONAL_T EVENT.

We provide the SYMBOLEO specification for our first example:

- **Initial Symboleo**: `O(WebMD, emdeon, true, not Happens(evt_notify))`
- **Updated Symboleo**: `Happens(evt_update_available) -> O(WebMD, emdeon, true, not Happens(evt_notify))`

Adding an Antecedent

All other cases of conditional keywords followed by an event will correspond to the addition of an antecedent, captured in the CONDITIONAL_A EVENT pattern. While there were many cases of refinements that resulted in this operation, many of them could belong to the same pattern class. 'If', 'In case', and 'In the event' are all very common conditionals that could be synonymous with each other, and therefore part of the same pattern class. Examples include:

- **In the event of EVENT**: *in the event of a force majeure*, the party affected thereby shall give immediate written notice to the other.
- **In case EVENT**: *In case developer terminates the agreement*, it shall hand over the entire project...
- **If EVENT**: You must sign a general release *if you renew your franchise.*

We had previously mentioned the ambiguity arising from synonyms for the word 'after', such as 'upon', 'once', and the word 'after' itself. We discussed that in the context of a *norm*, these patterns favour interpretation as a *conditional*. We can see that this is indeed the case in the following examples:

- **after EVENT**: The employer shall be the beneficiary of the remaining death proceeds of the policy *after the employee's interest is determined.*
- **Upon EVENT**: *Upon approval by Cerus of the corrective plan*, Porex shall then make the corrections...
- **Once EVENT**: *Once the holding company is generating revenue*, the parties will negotiate a monthly fee.

In all of these examples, the adjuncts can be treated as conditionals. We can therefore add these keywords to the CONDITIONAL_A rule. The exception for this is when the keyword appears *outside* the context of the norm consequent, and in an existing conditional, as we saw in the `HappensAfter` discussion earlier. We provide the SYMBOLEO specification for our third example:

- **Initial Symboleo**: `O(you, owner, true, not Happens(evt_sign_release))`
- **Updated Symboleo**: `O(you, owner, Happens(evt_transfer_franchise), not Happens(evt_sign_release))`

Our mappings for all conditional statements are as follows:

- CONDITIONAL_T EVENT: Add trigger
- CONDITIONAL_A EVENT: Add conditional

We also note the impact of a conditional refinement on a *power to terminate the contract*, which appears quite often in our construction set (e.g., '*In the event the company is in default*, lessor may elect to terminate this agreement'). If we consider the norm without the conditional, we simply get 'lessor may elect to terminate this agreement'. If there were multiple such conditional powers on a contract, then the broadened contract template would consist of duplicate unconditional powers to terminate the contract, which is a scenario we want to avoid. Instead, we may want allow for a *special operation* that allows for the creation of this very specific power. This violates our principle of adding new norms, but since this refinement is ubiquitous and the consequent is a *predictable* power termination, we include it as a special case for both conditional patterns.

4.4.7 New Patterns and Operations

We have now addressed the semantics found in real contracts for all of the operations that directly involve a norm refinement. The NL refinements corresponding to these operations were found by looking for trigger words and patterns based on the heuristics from RQ2. However, in order to mitigate false negatives, we want to broaden the search and look for more general linguistic structures that correspond to our target Symboleo operations. To this end, we can leverage various NLP libraries, such as SpaCy. We saw an example in Fig. 2.5 of how SpaCy can parse out the linguistic structure of sentences. We can run our dataset through this NLP processing pipeline and extract generic prepositional phrases and subordinate clauses, both of which can function as adjuncts and have the possibility of being a source of unidentified NL refinement patterns. This broader search has revealed some new patterns and operations. Some of these require simple modifications to pattern classes and operations that have already been identified. Others are slightly more complicated and introduce new patterns, which we will now discuss.

Exception Statements

We found a cluster of refinements that deal with *exceptions* to existing norms, which can be framed in terms of temporal and conditional refinements. These are associated with keywords such as 'without', 'unless', and 'except', as seen in the following examples:

- **without EVENT**: The distributor shall not be entitled to engage sub-distributors for sales of the product, *without having obtained Lucid's prior written approval.*

- **unless EVENT**: *Unless approved by party a*, party b shall pay liquidated damages to party a at the standard rate of RMB 500 per trip.
- **except EVENT**: The Contractor agrees that they will not disclose any Confidential Information, *except as authorized by the Client*

Most of these are refinements are used to denote an exceptional case that deals with permission or authorization. The corresponding operation to which these map is somewhat more complicated than previous examples we have seen. Consider our first example. If we remove the refinement, we have a simple obligation *not* to perform an event. If we add the 'without' adjunct, which represents an exception, then the distributor *is* entitled to perform the event. Therefore, the corresponding operation is to cancel (*terminate*) the existing obligation. The second and third examples are similar. Without the refinements, we have basic obligations. Adding the refinements back in terminates these obligations. Consider this simplified example to illustrate:

- **T:** Party1 must perform Event X for Party2
- **C:** Party1 must perform Event X for Party2 *unless Event Y happens*
- **S(T):** `ob1: O(Party1, Party2, true, Happens(Event X))`
- **S(C):**
  ```
  ob1: O(Party1, Party2, true, Happens(Event X))
  pow_terminate_ob1: P(Party2, Party1, true, Terminated(ob1))
  ```

The 'unless' refinement results in the creation of a new power (pow_terminate_ob1), the exertion of which will result in the termination of an initial obligation (ob1). As we saw with our termination power exception, this would introduce another operation where we are creating a new norm. Like that previous operation, this operation involves adding a *predictable* power for a case that appears often in real contracts. We therefore choose to admit it into our CNL. We can capture this pattern as P_EXCEPT EVENT, where P_EXCEPT includes 'without', 'unless', and 'except'.

Notice of Termination

Another common refinement is the case where a party has the power to terminate the contract by providing notice of termination to the other party *in advance*, exemplified in the following norm: 'FIIOC may terminate this agreement at any time *upon ninety (90) days written notice to company*'. In SYMBOLEO this statement may be represented as follows: `Happens(evt_provide_notice) AND (CURRENT_DATE > Date.add(evt_provide_notice, 90, days) -> P(FIIOC, company, self, Terminate(self))`. Note the use of the conjunctive 'AND' as well as the date comparison. This is a type of propositional statement that we have not yet seen in this work, as we have been considering propositions *only* in the form of predicate functions. However, it *does* exist in the XText specification that we have been following. Due to the ubiquity of this scenario, we can introduce this special refinement operation. Essentially, we are confirming that the notice has been

provided, *and* that the specified timespan has elapsed since that notice. If this is the case, then the party has the power to terminate.

We also note that if we were to remove the refinement, we would once again have a simple power to terminate. We have already discussed how we want to avoid having unconditional contract termination powers in our template, and we introduced the special operation of adding a termination power to avoid this. Building on that idea, we can add another similar special operation with the added feature of including a timespan with a notice event. The pattern class that we associate with this scenario is CONDITIONAL_N TIMESPAN NOTICE_EVENT, where NOTICE_EVENT specifically refers to an event that involves giving notice of termination (e.g. 'termination notice from the seller)'. Once again, we allow for this special case because of its ubiquity. Based on examples in our dataset, the CONDITIONAL_N rule includes 'upon' and 'with'.

Rights and Permissions

We will now discuss one final potential refinement found in this analysis. We came across a handful of cases where a refined 'right' is expressed, such as in these examples:

- **from TIMEPOINT to TIMEPOINT**: Eligible PIN codes may be downloaded *from October 1, 2009 to March 31, 2010*.
- **until DATE**: licensee shall have the exclusive right and license to develop and distribute wireless products *until December 31, 2006*.
- **until EVENT**: Cremer may refuse to deliver the product to Ultragenyx *until the parties agreed on a respective price*.

Removing the refinements results in rights to perform some action, which need no representation in Symboleo. However, the introduction of the refinements create *exceptions* around these rights, some of which must be expressed in terms of obligations and powers. Consider our third example. Once the until EVENT refinement is introduced, the party *no longer* has the right to perform the action. This means they now have an obligation *not* to perform it, which would need to be created. We have previously violated our principle of adding norms in cases of adding powers, since the consequents are predictable and reference an existing obligation. This is not necessarily the case here, so we will *not* be including this operation in our CNL, and it may be left to future work.

4.4.8 Final Results

We now present the final list of mappings (Table 4.2) and the grammar (Listing 4.5) that comprise our CNL.

Listing 4.5: Final grammar

```
<EVENT> ::= <SUBJ> <VP>
<SUBJ> ::= simple_np
```

Table 4.2: Final CNL mappings

Pattern Class	**Symboleo Operation**
P_BEFORE_S DATE	Happens → ShappensBefore
until DATE *	Happens → ShappensBefore
within TIMESPAN P_AFTER_W EVENT	Happens → WhappensBefore
at least TIMESPAN P_BEFORE_T EVENT	Happens → WhappensBefore
at least TIMESPAN P_AFTER_T EVENT	Happens → WhappensBefore
P_BEFORE_E EVENT	Happens → WhappensBeforeEvent
until EVENT *	Happens → WhappensBeforeEvent
P_AFTER_E EVENT	Happens → WhappensBeforeEvent **
P_AFTER DATE	Happens → HappensAfter
P_AFTER_PF TIMESPAN P_AFTER_T EVENT	Happens → HappensAfter
P_AFTER_PF TIMESPAN P_BEFORE_T EVENT	Happens → HappensAfter
between TIMEPOINT and TIMEPOINT	Happens → HappensWithin
P_DURING TIME_PERIOD	Happens → HappensWithin
for TIMESPAN P_AFTER_I TIMEPOINT *	Happens → HappensWithin
P_EXCEPT EVENT	Add Obligation Termination Power
CONDITIONAL_A EVENT	Add Antecedent
CONDITIONAL_T EVENT	Add Trigger
CONDITIONAL_A EVENT	Add Contract Termination Power
CONDITIONAL_T EVENT	Add Contract Termination Power
CONDITIONAL_N TIMESPAN NOTICE_EVENT	Add Contract Termination Power

* Only applies to negated events
** Requires swapping arguments, and only applies to refining conditional statements

```
<VP> ::= [<ADVB>] (<TVP> | <LVP> | <IVP>) [<PP>]

<TVP> ::= transitive_verb <DOBJ>
<LVP> ::= linking_verb <PRED>
<IVP> ::= intransitive_verb

<ADVB> ::= adverb
<DOBJ> ::= simple_np
<PRED> ::= adjective
<PP> ::= preposition <POBJ>
<POBJ> ::= simple_np

<NOTICE_EVENT> ::= "termination notice from" <NOTIFIER>
<NOTIFIER> ::= (role from domain model)

<DATE> ::= (various date formats)
<TIMEPOINT> ::= <DATE> | <EVENT>
```

```
<TIMESPAN> ::= <TIME_VALUE> <TIME_UNIT>
<TIME_VALUE> ::= Number
<TIME_UNIT> ::= "hours" | "days" | "weeks" | ...

<P_BEFORE_S> ::= "before" | "by"
<P_BEFORE_T> ::= "before" | "prior to"
<P_BEFORE_E> ::= "before" | "prior to"

<P_AFTER_W> ::= "after" | "following" | "from" | "of"
<P_AFTER_T> ::= "after"| "following" | "from"
<P_AFTER_E> ::= "after"
<P_AFTER> ::= "after" | "later than"
<P_AFTER_PF> ::= "following"
<P_AFTER_I> ::= "following" | "after"

<P_DURING> ::= "during" | "throughout" | "within"
<P_EXCEPT> ::= "without" | "unless" | "except"

<CONDITIONAL_T> ::= "when"
<CONDITIONAL_A> ::= "after" | "if" | "in the event [of]" | "in
   case" | "once" | "upon"
<CONDITIONAL_N> ::= "upon" | "with"
```

We began this chapter by situating our problem as a templating approach with a goal of deciding what format the template parameters will take. RQ1 involved listing all possible operations that could be done to a SYMBOLEO contract, and deciding which of these operations we are interested in. We did this by following the principle that we only want to include operations that will *refine* the contract. This was primarily done with a *syntactic* analysis of SYMBOLEO. Then, in RQ2, we analyzed these operations from a *semantic* perspective, finding general semantic patterns that correspond to each operation. We introduced the principle of integrity, which ensures that we are always working with a grammatically correct NL contract and corresponding valid SYMBOLEO specification. We aligned our operations with the concept of the linguistic adjunct, and came up with a set of heuristics representing a working CNL that consisted of an initial set of mappings and a corresponding grammar. Finally, here in RQ3, we refined this CNL by incorporating these heuristics into an evidence-based approach to constructing a more precise CNL.

4.4.9 Precise Formulation of Problem

Before moving on to our evaluation of the CNL, we will make some of the terms we have used throughout this thesis more formal. We have mentioned the concepts of T, C, S(C), and S(T) throughout this work, but we have not yet formally defined T and C. We are now in a position to do so.

We have been referring to T as the contract template. More formally, it is a list of n parameters t_i: $T = \{t_1, t_2, \ldots, t_n\}$. Each t_i consists of a NL sentence n_i with a collection of m parameters denoted by P_i: $t_i = (nl_i, P_i)$, $P_i = \{parm_{i1}, parm_{i2}, \ldots, parm_{im}\}$. In most

cases, $m \in \{1, 2\}$, meaning each sentence has 1 or 2 parameters. For example, we may have the template sentence n_k: '[P1] The seller must deliver the goods to the buyer [P2]'. There is potential for a parameter to go on either side of the original NL, so $m = 2$. Each parameter $parm_{ij}$ for a templated sentence n_i is associated with a corresponding norm id from the SYMBOLEO contract: $parm_{ij} \in \{obligations.o_1, obligations.o_2, ...\}$. In this way, the NL sentences are linked to the SYMBOLEO specification. Here we re-emphasize the assumption that we have a correspondence between a single sentence and a norm, which is not always the case in legal contracts.

Now let us specify the CNL more precisely. Our CNL consists of a set of a mappings $CNL = \{mapping_1, mapping_2, ..., mapping_a\}$. Each mapping $mapping_i$ consists of a pattern class pc_i and an operation op_i: $mapping_i = (op_i, pc_i)$ For example, we have a mapping $mapping_2$, where $pc_2 =$ CONDITIONAL_A EVENT and $op_2 =$ 'Add an antecedent'.

C is the customized contract, and can be represented as follows: $C = \{c_1, c_2, ..., c_n\}$, where c_i corresponds to a CNL refinement on t_i. Each parameter $parm_{ij} \in P_i$, will be replaced by a refinement r_{ij}, which corresponds to some pattern class pc_k from our CNL mappings, denoted by $r_{ij} \rightarrow pc_k$:

$c_i = (n_i, R_i)$
$R_i = \{r_{i1}, r_{i2}, \ldots, r_{im}\}$
$r_{ij} \rightarrow pc_k$
$m_k = (op_k, pc_k)$
$m_k \in CNL$

We will refer to this precise formulation of our problem sparingly, but we include for completeness. We can now move on the evaluation of our CNL.

4.5 Evaluation

We mentioned that the main metric of interest for our CNL construction was *pattern coverage*. This metric poses the question: Given a norm refinement from an arbitrary NL contract, can this refinement be expressed in our CNL grammar, *and* does it map to our predicted SYMBOLEO operation? To measure this, we split off a test set using 20% of our dataset, which represents about 63 refinements, each of which has a corresponding SYMBOLEO operation associated with it. Values in this test set were specifically *not* used in the construction of the CNL and were reserved for this specific purpose of measuring pattern coverage.

We must repeat a few steps from the CNL construction on the test set. First, we need to find the pattern within each refinement by abstracting away the primitive concepts (EVENT, DATE, TIMESPAN, etc.). We will then have a set of pairings of patterns and operations. For each of these pairings, we will verify two properties:

- Does the pattern expressed in the refinement belong to one of the existing pattern classes from our constructed CNL?
- If it does, then does the operation in the pairing match the operation in our CNL?

Referring back to our precise formulation of our problem, suppose now that we have a pairing of a pattern and operation (p_i, op_i). We want to know if p_i corresponds to any of the pattern classes $pc_k \in CNL$. If it does, then does $op_i = op_k$? For example, suppose we have the norm 'Party A must perform event Z in the event Party B performs event W'. The refinement r_i is 'in the event Party B performs event W'. The pattern of this refinement p_i is in the event EVENT, which corresponds to the pattern class CONDITIONAL_A EVENT, which we will call pc_j. Therefore, $p_i \rightarrow pc_j$. We manually verify the SYMBOLEO operation op_i that this refinement corresponds to, and it turns out to involve adding an antecedent, which is equal to op_j. This refinement is therefore *covered* by our existing CNL.

Any refinements that fail this criterion will be considered a miss, and would suggest that our CNL needs to be further improved. We are interested in counting the number of misses found in our test set, giving us a simple metric to represent pattern coverage. On its own, a single number is not very explanatory, so we will analyze specific examples from the test set.

4.5.1 Evaluation Examples

Perfect Matches

Examples of test set members that satisfy *both* conditions, along with their general patterns (bold), include:

1. **within TIMESPAN of EVENT**: Payment will be made by reseller to company *within 5 days of receipt of payment.*
2. **unless EVENT**: Licensor shall make a first delivery of content to plan_b *unless separately agreed between the parties.*

The first example aligns with our within TIMESPAN P_AFTER_W EVENT pattern class, and the refinement corresponds to a `Happens` $\rightarrow$ `WhappensBefore` refinement. The second example aligns with the P_EXCEPT EVENT pattern class and corresponds to the suspension of an existing obligation. In both cases, the pattern class is one already identified in our CNL construction, *and* the operation is the one predicted by our constructed CNL. If all of the cases in our test set met these criteria, then we would consider this 100% pattern coverage.

Close Matches

Next, consider the following examples from the test set:

1. **provided that EVENT**: The Contractor will be entitled to pro rata payment of the Compensation *provided that there has been no breach of contract on the part of the Contractor.*

2. **on EVENT**: Client shall make a prepayment of $1,900 and pay the remaining $3,100 *on completion of the scope of work*.

In both of these examples, we have *new* general patterns that were *not* seen in the construction set. However, in both cases, the general patterns can be included in a pattern class that has *already been identified*, namely the CONDITIONAL_A EVENT. This type of value in the test set is not considered a *miss*, but it does suggest that our rules could be refined to include more keywords. In this case, it suggests we might want to add 'on' and 'provided [that]' to our CONDITIONAL_A rule. We would only consider a test value to be a *miss* if it required a new *pattern class*.

Missing Pattern Class

We find two 'misses' in our test set, each of which suggested a new mapping for our CNL:

1. ...the Seller may suspend performance of all of its obligations under the agreement *until payment of amounts owed has been received in full*
2. *from the effective date*, consultant shall not solicit any employee to terminate employment...

In the first example, we have a power to suspend an obligation. Introducing the until EVENT refinement suggests this power is no longer in effect once the EVENT occurs. The closest mapping from our constructed CNL is the pattern class P_EXCEPT EVENT, which corresponds to the operation of *suspending an obligation*. There are two important things to note here. First, the keyword 'until' acts very much like our P_EXCEPT keywords (unless, except, without). Therefore, this example suggests that we add this keyword to the P_EXCEPT rule. The second point is that we are initially dealing with a *power*, rather than an obligation, and this is not accounted for in our CNL mapping. Since the initial power *already* involves a suspension, the impact of the refinement is to essentially 'suspend' the suspension. Symboleo has a provision for this – namely the special event of *resuming* an obligation. This also suggests, that in general, the context of the initial norm is important when dealing with these exception refinements. We also note that this violates our principle of adding new norms, but once again, the new norm has a predictable consequent that references an *existing* norm, which may be grounds for including it in a future iteration of the CNL.

In our second example, removing the refinement gives us an obligation *not* to perform an event. The refinement could be said to follow a from TIMEPOINT pattern and the resulting Symboleo operation is to change the `not Happens` predicate to a `not HappensAfter` predicate. It is somewhat similar to our after EVENT pattern class from our CNL, but the semantics are somewhat different, in that we would not be able to always substitute 'from' for 'after' in these refinements. We can consider the from TIMEPOINT analogous to the until TIMEPOINT pattern, in that both patterns suggest that they are refining *situations* rather than events. While the until TIMEPOINT will introduce an *ending* point for a situation, the from TIMEPOINT introduces a *starting* point. In our initial semantic analysis in RQ2,

we posited the notion of a from TIMEPOINT until TIMEPOINT, and here we have a case where the from TIMEPOINT is on its own.

These are the only two examples that we find where the test pairing of a NL refinement and Symboleo operation were *not* represented in our constructed CNL. We also found a few more cases of *rights* that included refinements, but these would lead to *new obligations* as previously discussed. We do not include this type of operation in our CNL, but we note that the pattern classes used to express these refinements were also found in our construction set. We found *no cases* where we encountered a refinement that was included in an existing pattern class but mapped to a *different* operation than the one specified in our CNL. While we cannot rule out the possibility of this ever happening (given a larger dataset for example), this initial result suggests a certain degree of uniformity in how these patterns are used in real contracts.

4.5.2 Analysis and Threats to Validity

Out of our 63 test values, we found 5 'close matches' that suggested adding additional keywords to existing pattern class rules (e.g. adding 'provided that' to our CONDITIONAL_A rule. We do not consider these misses, but they are worth mentioning. We found only 2 'missed' examples, giving us a pattern coverage of 61/63 = 0.97.

We now note some key limitations and threats to our construction and evaluation. In the previous section, we discussed the analysis process for constructing a CNL from a dataset, and here we will address potential threats associated with each of those steps. First, we re-emphasize that this analysis would become more reliable with a larger dataset that had more expert input. We filtered the CUAD dataset to consider only norms from smaller contracts. As related work in the Symboleo ecosystem progresses, more contracts from the entire dataset could be incorporated into the test set. These new contracts may be mixed in to the entire dataset before splitting off the test set (as we have done in this iteration), or we may consider applying our pattern coverage evaluation to an entirely new contract. This latter approach would avoid polluting the construction set with data from a contract in the test set. While the CUAD dataset includes contracts from a variety of industries, we note the bias that all of these come from the same database (EDGAR), and therefore include the property that they must be shared publicly with the SEC. There may be the possibility that contracts that do *not* need to be shared with the SEC may have different linguistic character. Therefore, in addition to incorporating more contracts from CUAD, there may be benefits to incorporating completely separate *datasets* from other sources. We used contracts from a variety of industries, but in the future we may identify specific industries where Symboleo is most useful. Further analysis might then consist in analyzing contracts *specifically* from those more specialized industries. In this dataset, I was the only annotator for the dataset, and reliability would be greatly improved with further input from experts in Symboleo and/or the legal domain. Related work, such as GaiusT [53], where inter-annotator agreement is emphasized, offers an example that could be followed.

Due to the imprecision and ambiguity inherent in NL, this analysis has necessarily involved the use of various heuristics, and though we made efforts to ensure these heuristics

were well-informed (through careful linguistic analysis and building on relevant related work), we note that this can introduce further threats to validity, especially with regards to false negatives. In our initial search for norms, we used a heuristic of normative keywords that were used in much of the related work we studied. However, it is very possible that we missed many norms, and therefore norm *refinements*, that were not captured by these heuristics. Once we have identified a set of norms, there is further opportunity for false negatives in cases where the refinement is not captured by our linguistic refinement heuristics introduced in RQ2. The goal of RQ2 was to come up with a strong linguistic foundation for these heuristics. Indeed, our constructed CNL from RQ3 was in fact quite similar to the working CNL introduced in RQ2, which was based on a *general* semantic analysis (rather than one specifically focused on contracts). While this can be seen as validation of our efforts in RQ2, we acknowledge that there may still be some refinements in norms that use *other* linguistic structures or perhaps even metaphorical language (though this is less likely in the legal domain) that were completely missed in our analysis. We attempted to mitigate this by performing a *broader* search for relevant linguistic structures, but there is still room for false negatives. Supposing further research identifies such false negatives, this might result in new pattern classes on our CNL, or additional mappings between existing pattern classes and existing operations.

While the linguistic heuristics found in RQ2 help mitigate missed *patterns*, the work done in RQ1 of exhaustively listing all Symboleo operations helps mitigate missed *operations*. There is room for error here as well, as our evaluation illustrated the possibility of new operations that were not considered in RQ1, namely the operation of adding powers that reference *existing* norms. We had initially discounted this operation on a principle of disallowing new norms, but our analysis showed that these ubiquitous operations were predictable enough to be admitted. Our analysis also involved considering *rights* that are expressed in NL contracts, since they may have an impact on the Symboleo specification. Our analysis found that any relevant refinements on existing rights resulted in the operation of creating a new obligation. We have excluded this operation for now, but this is another area that could be further explored in future work.

We also note the importance of specifying this analysis *methodology*. While we have generated an evidence-based CNL, these threats highlight how the CNL may have room for improvement. By introducing this formal and partially-automated methodology, we have a means for iterating on this CNL in future work. Future work may also involve automating further steps in the construction process.

Future Metric: Event Specification Coverage

Finally, we discuss the potential metric of *event specification coverage*, which may be a useful metric to quantify in future work. In this work, we have largely abstracted away the details of events, focusing instead on the general refinement patterns. However, in order for this work to be fully applied to real contracts, we want to be able to capture the variety of events that are specified in real contracts. There may be events that we can capture *synonymously* using our CNL event specification.

Consider once again the ‘receiving’ event found in the following refined norm: ‘Dolphin agrees to complete its photo-editing services within 14 days of *receiving the original digital*

photo files'. Our current event specification requires a subject, so we could capture the event as *Dolphin receiving the original digital photo files.* Since our captured event is synonymous with the original event, we could consider this event as being covered by our current event specification. A more complex event, such as one that uses a variety of conjunctions and disjunctions, would *not* be covered. A formal analysis of the event specification coverage would tell us how likely it is that a real contract event could be covered by our event specification *and* it would provide us with more concrete ideas on how to *improve* the event specification in our CNL.

4.6 Summary

To summarize, we have defined a 3-step methodology for creating an evidence-based CNL for customizations used for the formalization of legal contracts into Symboleo. The first step (RQ1) involves defining the range of the output restricting the full range of possible operations using set of guiding principles. For each potential operation defined, we use linguistic analysis to come up with a set of potential NL refinements that result in each operation, resulting in a working CNL (RQ2). We then analyze a real contract dataset by applying this working CNL and other NLP-based heuristics to identify refinement patterns and corresponding operations found in real contracts, resulting in our final CNL (RQ3). We measure our coverage of the CNL by splitting off a test set and verifying that each refinement can be expressed in our CNL and that the refinement maps to the predicted operation. Now that we have an evidence-based CNL with well-established pattern coverage, we are ready to discuss our SymboleoNLP tool, which will encode this CNL, and help prove our concept of automated refinement formalization.

Chapter 5

Application

In this chapter, we introduce the tool SymboleoNLP, which will demonstrate the concept of automated contract refinement formalization using our CNL. We will first state the problem more clearly with a concrete example and discuss the key requirements, which are directly related to our CNL (Section 5.1). We will then describe in detail how these requirements are fulfilled, by building on our example and providing code samples. In particular:

- Section 5.2 addresses the first requirement, which is to ensure that valid CNL is entered into the system.
- Section 5.3 focuses on the second requirement, which is to properly map CNL constructs to a predictable set of Symboleo refinement operations.

The first norm of a sample rental contract is used to illustrate the process in the above sections, whereas Section 5.4 walks through the process using the remaining norms of the example. The source code for SymboleoNLP is publicly available on GitHub [1].

5.1 Problem Statement

Given a NL contract template and its corresponding Symboleo specification, SymboleoNLP will allow a contract author to refine the NL contract using the pre-defined set of pattern classes found in our CNL. This refinement will update the underlying Symboleo specification in a predictable way by using the Symboleo operation to which the pattern class maps. The resulting artifacts include a refined NL contract that is understandable by the contract author and other interested parties, as well as a Symboleo specification that can be used with existing model-checking and code-generation tools. There are two important points we must emphasize from this description. First of all, the parameters are *optional*. By our principle of integrity introduced in Section 4.3, both the NL contract and the Symboleo specification are guaranteed to be valid (albeit broad) at any stage of

[1] `https://github.com/reganmeloche/symboleo-nlp`

the refinement, including the beginning. Secondly, the initial template in question must conform to our standards. We cannot simply take an arbitrary contract or contract template and directly use it with SymboleoNLP. The parameters (blanks) in the template must first be made usable with our CNL. This is a manual process which we will now walk through using an example.

5.1.1 Manual Preparation of the Template

We will begin with an arbitrary NL contract `T0`. We make very few assumptions about this contract. It may be a complete contract, or it may be a traditional FITB template. Let's suppose we have a simple NL contract as shown in Table 5.1. The first step is to manually convert this to a Symboleo specification, including the domain model, declarations, and contract specification. We will call this `S(T0)`. This is presented in Listing 5.1.

The renter must pay a security deposit of $1000 within 2 weeks of occupying the premises. The renter may not keep any pets on the property unless authorization is provided by the landlord. In the event of a late payment, the renter must pay an extra fee of $50.

Table 5.1: Simplified Rental Agreement Sample (T0)

Listing 5.1: Symboleo specification of the rental agreement S(T0)

```
1  Domain rentalDomain
2    Renter isA Role with name: String;
3    Landlord isA Role with name: String;
4    Property isAn Asset with address: String;
5    Pets isAn Asset;
6    SecurityDeposit isAn Asset with amount: Number;
7    PayDeposit isAn Event with payer: Role, payee: Role, deposit: SecurityDeposit;
8    OccupyProperty isAn Event with property: Property, occupier: Role;
9    KeepPets isAn Event with keeper: Role, pets: Pets, property: Property;
10   AuthorizePets isAn Event with grantor: Role, allowance: Pets;
11   PayExtra isAn Event with payer: Role, payee: Role, amount: Number;
12 endDomain
13
14 Contract RentalAgreement (
15   renter_name: String,
16   landlord_name: String,
17   property_address: String,
18   deposit_amount: Number,
19   extra_fee_amount: Number
20 )
21
22 Declarations
23   renter: Renter with name := renter_name;
24   landlord: Landlord with name := landlord_name;
25   the_property: Property with address := property_address;
26   pets: Pets;
27   security_deposit: SecurityDeposit with amount := deposit_amount;
28   evt_pay_deposit: PayDeposit with payer := renter,
29     payee := landlord, deposit := security_deposit;
```

```
30    evt_occupy_property: OccupyProperty with property := the_property,
31      occupier := renter;
32    evt_keep_pets: KeepPets with keeper := renter,
33      pets := pets, property := the_property;
34    evt_authorize_pets: AuthorizePets with grantor := landlord,
35      allowance := pets;
36    evt_pay_extra: PayExtra with payer := renter,
37      payee := landlord, amount := extra_fee_amount;
38
39  Obligations
40    ob_pay_deposit: O(renter, landlord, true, WhappensBefore(evt_pay_deposit, Date.add(
        evt_occupy_property, 2, weeks)));
41    ob_no_pets: O(renter, landlord, true, not Happens(evt_keep_pets));
42    ob_pay_extra: O(renter, landlord, Happens(Violated(obligations.ob_pay_deposit)),
        Happens(evt_pay_extra));
43
44  Powers
45    pow_allow_pets: Happens(evt_authorize_pets) -> P(renter, landlord, true, Terminated(
        obligations.ob_no_pets));
46  endContract
```

The next step is to *broaden* the contract. For any norms that are *refined* in S(T0), we want to convert these to their more broadened forms by reversing any operations identified in our CNL. For example, our contract contains a `WhappensBefore` predicate, which we can broaden into a `Happens`. If a norm has a trigger or antecedent, then we can broaden the norm by removing those as well. Most *broadened* norms will therefore have no trigger, a 'true' antecedent, and a simple `Happens` predicate as the consequent. A special case is where we might have an initial contract with a *refined conditional* (e.g., 'If event X happens *before* DATE'). Since this is a manual process, there will be some flexibility in how these cases are handled. They could be broadened to remove the refinement ('If event X happens'), or they could be removed completely. In the latter case, it is worth noting that the original would *not* be able to be re-created by our pattern classes, so in general we opt for the former approach. We saw certain cases where the refinement operation is different depending on whether it appears in a conditional or the consequent. For example, if we have a refinement of the form after EVENT, that could refer to the COND_A EVENT or the P_AFTER_E EVENT pattern class, each of which maps to a different operation. In these ambiguous cases, further context about the parameter being refined will be required to resolve the correct pattern class. Finally, also note that the broadened form of the contract removes the power (pow_allow_pets) completely. In our CNL construction, we have discussed methods for re-introducing powers with predictable consequents such as this through refinements, which we will see applied in this chapter.

Once we have broadened the norms in the contract specification, we want to look for any corresponding domain model objects and declarations that are no longer used in the broadened contract specification. For example, by broadening the `WhappensBefore` predicate, we have removed any mention of the evt_occupy_property declaration. We must therefore remove that declaration from the specification as well as the corresponding domain event OccupyProperty. We now have a fully-broadened contract, which we will call S(T), which is shown in Listing 5.2.

Listing 5.2: SYMBOLEO specification of the broadened rental agreement S(T)

```
1  Domain rentalDomain
2    Renter isA Role with name: String;
3    Landlord isA Role with name: String;
4    Property isAn Asset with address: String;
```

```
  Pets isAn Asset;
  SecurityDeposit isAn Asset with amount: Number;
  PayDeposit isAn Event with payer: Role, payee: Role, deposit: SecurityDeposit;
  KeepPets isAn Event with keeper: Role, pets: Pets, property: Property;
  PayExtra isAn Event with payer: Role, payee: Role, amount: Number;
endDomain

Contract RentalAgreement (
  renter_name: String,
  landlord_name: String,
  property_address: String,
  deposit_amount: Number,
  extra_fee_amount: Number
)

Declarations
  renter: Renter with name := renter_name;
  landlord: Landlord with name := landlord_name;
  the_property: Property with address := property_address;
  pets: Pets;
  security_deposit: SecurityDeposit with amount := deposit_amount;
  evt_pay_deposit: PayDeposit with payer := renter,
    payee := landlord, deposit := security_deposit;
  evt_keep_pets: KeepPets with keeper := renter,
    pets := pets, property := the_property;
  evt_pay_extra: PayExtra with payer := renter,
    payee := landlord, amount := extra_fee_amount;

Obligations
  ob_pay_deposit: O(renter, landlord, true, Happens(evt_pay_deposit));
  ob_no_pets: O(renter, landlord, true, not Happens(evt_keep_pets));
  ob_pay_extra: O(renter, landlord, true, Happens(evt_pay_extra));

Powers

endContract
```

Finally, we must broaden the NL contract accordingly. This involves isolating the specific adjuncts that were responsible for each norm refinement that we broadened, and replacing it with a parameter. For example, the broadening of the `WhappensBefore` to the `Happens` corresponds to removing '`within 2 weeks of occupying the property`' from our first sentence. The result is a NL contract template `T`, shown in Table 5.2 that can now be used with the tool.

pay_deposit: The renter must pay a security deposit of $1000 [P1]
no_pets: The renter may not keep any pets on the property [P2]
pay_extra: [P3] the renter must pay an extra fee of $50.

Table 5.2: Broadened Rental Agreement Sample (T) with parameters

We also add string identifiers to each sentence. These will be used by the contract authors for referencing specific sentences in the contract, as well as for linking the NL statements to norm in the Symboleo specification. We re-iterate that in many cases, the correspondence between the NL and the Symboleo refinement will not be so straightforward, and acknowledge this as a limitation. The process we have just described is entirely manual, though future work could potentially automate some aspects of it. For

example, supposing we have manually created `S(T0)`, we could automatically identify the broadening operations required to create `S(T)`, as well as the adjuncts that correspond to those broadening operations.

SymboleoNLP must now be loaded with our CNL, the NL template `T` and the corresponding Symboleo specification `S(T)`. The user will be given the option of refining any of the three parameters (P1, P2, P3) that are now present in `T`. They would select a parameter to customize, and enter a refinement using the patterns prescribed by the CNL. When the user completes the refinement on a parameter, SymboleoNLP will process the input and perform the corresponding operations to the Symboleo specification. Once the user has filled in all of the desired parameters, the contract and the corresponding specification can be saved for further use as needed. The entire process is shown in Fig. 5.1.

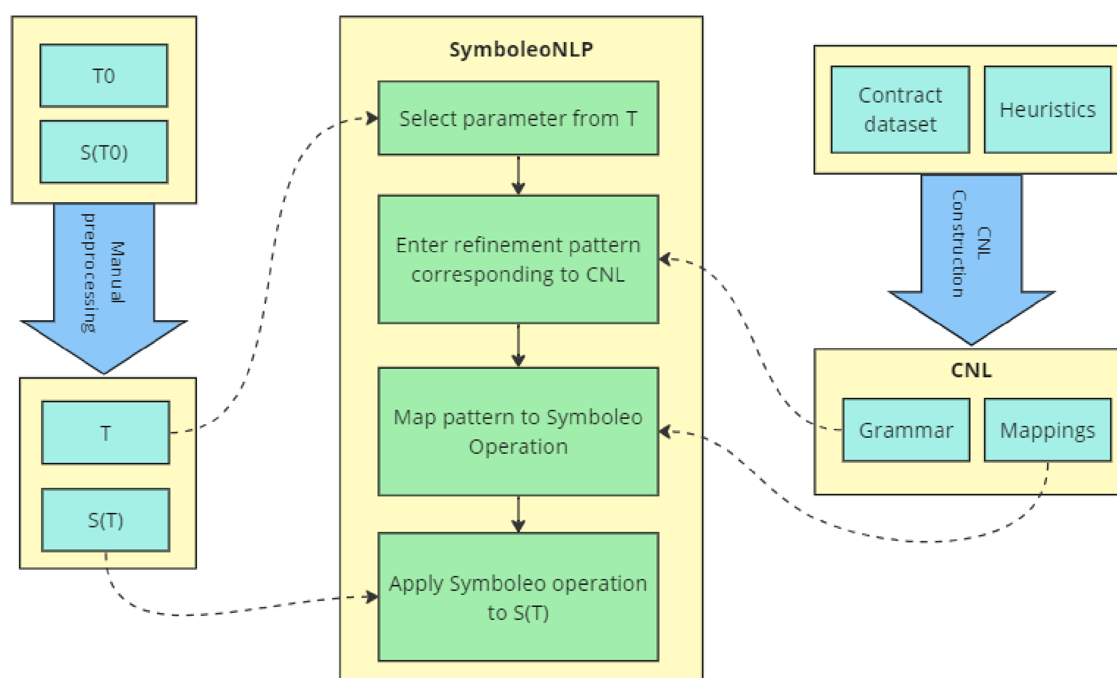


Figure 5.1: Diagram showing inputs and flow of SymboleoNLP

There is another question concerning our parameters that we must address before discussing the requirements in more detail. Given a parameter on a contract template, do we want to restrict the *types* of refinements allowed? For example, in our sample contract, we broadened 'The renter must pay a security deposit of $1000 within 2 weeks of occupying the premises' to 'The renter must pay a security deposit of $1000 [P1]'. In the original contract, P1 refers to a predicate refinement. However, it is possible that it could *also* be used for *other types* of refinements. We could put a conditional 'if' refinement or an exceptional 'unless' refinement in the parameter. All of these would be semantically correct and adhere to the CNL. However, there are cases where certain parameters will only work with certain types of refinements. For example, the until DATE refinement would not work semantically with a non-negated `Happens` predicate, since the word 'until' only makes sense with a *situation* or a *negated* event. In keeping with our goal of maximizing

flexibility, we will tentatively aim to allow parameters to work with *any* valid refinements, but we note there will be certain cases where this will need to be restricted. This discussion recalls the closely-related work of Tateishi et al. [88], where each parameter had its own unique refinement pattern. In our case, we have increased the flexibility by introducing a CNL that can be used for a large variety of refinements, though there are still certain limits on this flexibility.

5.1.2 Application Requirements

We identify the two key processes performed by the tool, which can be framed as our requirements and relate directly to our remaining RQs:

1. The user inputs a NL refinement that corresponds to our CNL. More formally: For any text that is input into the tool, it must correspond to one of the pattern classes defined in our CNL. Furthermore, *all* pattern classes should be included in the system. This requirement aligns with RQ4.

2. The refinement is mapped to the proper set of Symboleo operations. More formally, given a refinement that corresponds to a CNL pattern class, it must map to the set of Symboleo operations defined by our CNL. This requirement aligns with RQ5.

Let us consider our first contract obligation from Listing 5.2 as an example: 'The renter must pay a security deposit of $1000 [P1]'. Suppose we want to refine it to something synonymous with the text in the original contract. The refinement on the original contract is '`within 2 weeks of occupying the property`', which contains the event 'occupying the property'. This event has an implied subject (the *renter* is the occupying agent). Our event specification requires an explicit subject, so the refinement may take the following form: '`within 2 weeks of the renter occupying the property`'. This statement is allowed by our grammar and it adheres to the within TIMESPAN P_AFTER_W EVENT pattern class. Our first requirement ensures that the user can enter this refinement into the system. In fact, we would like it to guarantee that it *only* allows input that corresponds to our CNL pattern classes, and also that all pattern classes are covered.

Our second requirement is to ensure that this pattern class expression maps to the operation of refining the `Happens` predicate to the `WhappensBefore` predicate, which is defined in our CNL. Included in this requirement is that any new events introduced in the refinement are added to the domain model and declarations. For example, we would require that the `OccupyProperty` event is added to the domain model as well as a corresponding declaration (evt_occupy_property). This will result in an updated T and S(T). These three operations (refinement of the norm, adding the domain class, adding the declaration) are all applied to our Symboleo contract S(T). Once this is complete, the user can select another parameter for further customization. Our two requirements each contain some important sub-tasks. We will discuss the design and implementation of the tool with respect to these requirements and their sub-tasks, as well as their evaluation criteria. First, we discuss a few higher-level design decisions for the tool.

5.1.3 Tool Design Decisions

The language of choice for SymboleoNLP is Python. There are several motivating factors behind this decision:

- It is an easy-to-learn language. This lowers the barrier-to-entry for other developers who may need to use this project.
- Many existing NLP libraries, such as nltk and SpaCy, are implemented with Python. This makes it very easy to integrate with these technologies now and in the future.
- It has support for full-stack development, including database access, web APIs, and front-end web projects.
- It is a popular language with a large community. This makes it easier to find resources for debugging and problem-solving.

One of the first needs that our tool must address is the specification of Symboleo in Python. All of the code for processing the user's input will be done in Python, but ultimately we need to generate a valid Symboleo contract as an artifact. Therefore, we need to be able to represent a Symboleo contract using Python classes, and then convert this representation *back* into a valid Symboleo contract once we are done processing. We use our XText specification as a guide, but we note that there are a few important decisions where we strayed from this specification. We have made many internal cosmetic changes, for example changing some class names, to make it easier to work with using Python. These types of changes have no bearing on what we are able to ultimately represent in Symboleo. There are a few changes, however, that are more structural in nature. These changes add restrictions to what can be represented using our implementation. We now offer an illustrative example. Fig. 5.2 shows a snippet of the XText specification that describes a time Point object in Symboleo.

```
Point:
	pointExpression=PointExpression;

PointExpression:
	PointFunction |
	PointAtom;

PointFunction returns PointExpression:
	{PointFunction} name=PointFunctionName '(' arg=PointExpression ',' value=Timevalue ',' timeUnit=TimeUnit ')';

PointFunctionName:
	'Date.add';
```

Figure 5.2: Snippet of Symboleo's XText grammar showing the PointFunction

A Point consists of a PointExpression, which can take the form of a PointFunction or a PointAtom. A PointFunction then accepts an argument that is of the type PointExpression. This creates a circular dependency in our specification, which can lead to issues when parsing and constructing the objects in Python. We simply change the PointExpression

argument in the PointFunction to be a PointAtom instead. The only impact that has is that we cannot create a PointFunction inside another PointFunction, which would correspond to a relative date *within another* relative date. For example, this might look like 'within 2 weeks of 3 months of the contract terminating'. While NL indeed allows for this possibility, we found no cases of this type of statement in the contracts we analyzed, so we determine this to be safe change.

The use of Python for SYMBOLEO allows for the potential to more easily integrate useful Python libraries to our processing, and the Python specification can therefore be considered a useful contribution on its own, as it could be used in future applications in the SYMBOLEO ecosystem. This usefulness and flexibility comes at the cost of simplicity, as representing these structures in Python is quite complex (this partly motivates the use of a more elegant specification language like SYMBOLEO in the first place). To highlight this trade-off, we can compare a simple obligation in SYMBOLEO (`test_id: O(seller, buyer, true, Happens(evt_delivery))`) with the same obligation expressed in Python, in Listing 5.3. In the code, we introduce convenience methods for more efficient construction of these complex objects.

Listing 5.3: SYMBOLEO obligation expressed in Python

```
obligation = Obligation(
    id = 'test_id',
    trigger = None,
    debtor = 'seller',
    creditor = 'buyer',
    antecedent = Proposition(
        p_ands = [PAnd(
            p_eqs = [PEquality(
                curr = PComparison(
                    PNegAtom(
                        atom = PAtomPredicateTrueLiteral(),
                        negation = False
                    )
                )
            )]
        )]
    ),
    consequent = Proposition(
        p_ands = [PAnd(
            p_eqs= [PEquality(
                curr = PComparison(
                    PNegAtom(
                        atom = PAtomPredicate(
                            PredicateFunctionHappens(
                                VariableEvent('evt_delivery')
                            )
                        ),
                        negation = False
                    )
                )
            )]
        )]
    )
)
```

Before proceeding with our discussion on the requirements, we note that we are not overly concerned with the specifics of the user interface (UI). Our main goal is to show how a user can enter a NL pattern into a contract template, which will correspond to a predictable refinement operation in SYMBOLEO, resulting in a valid SYMBOLEO contract. Intuitive user design is crucial for adoption once the concept is proven, but there are

entire fields of research devoted to this. UI considerations are relevant for future work that might test if the partially-automated tool is capable of improving on a fully manual approach. For now, however, we are content with a simple web-based UI, allowing us to focus our efforts on the core functionality.

5.2 Generating CNL Patterns

Our first requirement is to ensure that valid CNL is entered into the system. We break this down into two key sub-tasks:

1. The user can *only* enter refinements that correspond to one of the pattern classes defined in our CNL.
2. It must be possible to enter *every* pattern class variation defined in our CNL as input into the system.

5.2.1 A Simpler Example

To illustrate how we approach this requirement, let us consider a much simpler CNL, consisting of the following pattern classes and corresponding EBNF grammar:

- P_BEFORE_X TIME_X
- P_BEFORE_X EVENT_X
- P_CONDITIONAL_X EVENT_X

Listing 5.4: Simple CNL grammar

```
<P_BEFORE_X> ::= "before"
<P_CONDITIONAL_X> ::= "if" | "in the event that"
<TIME_X> ::= "noon"
<EVENT_X> ::= <SUBJ> <VERB> <OBJ>
<SUBJ> ::= "alice" | "bob"
<VERB> ::= "buys" | "eats"
<OBJ> ::= "apple pie" | "a mattress"
```

At this stage, we are not concerned about corresponding Symboleo operations, as this will be addressed in the second requirement. All we need to know is that these are the pattern classes in our (simplified) CNL and they correspond to *some* pre-defined Symboleo operations. With this simplified CNL, it is not difficult to come up with *all* possible generations of each pattern class. Some of these include:

- 'before noon'
- 'before bob buys a mattress'

- ‘if alice eats apple pie’
- ‘in the event that bob eats a mattress’

We can calculate the total number of possible expressions using simple combinatorics on the terminal symbols in the grammar:

- P_BEFORE_X TIME_X has only 1 possible expression (‘before noon”)
- P_BEFORE_X EVENT_X has 8 possible expressions
- P_CONDITIONAL_X EVENT_X has 16 possible expressions

This gives us a total of 25 possible valid expressions in this CNL. Our goal then is to ensure that the user can *only* enter input that conforms to these expressions. We can achieve this by representing our CNL as a *directed graph*, where the nodes on the graph correspond to the symbols in our grammar. This approach to a graphical representation of a CNL was noted as an important feature by Wyner et al. [95]. A node A is connected to another node B if B follows A in any of the patterns. We will also require a common root node at the beginning, which means this graph is actually a *tree*. A version of this tree is presented in Fig. 5.3.

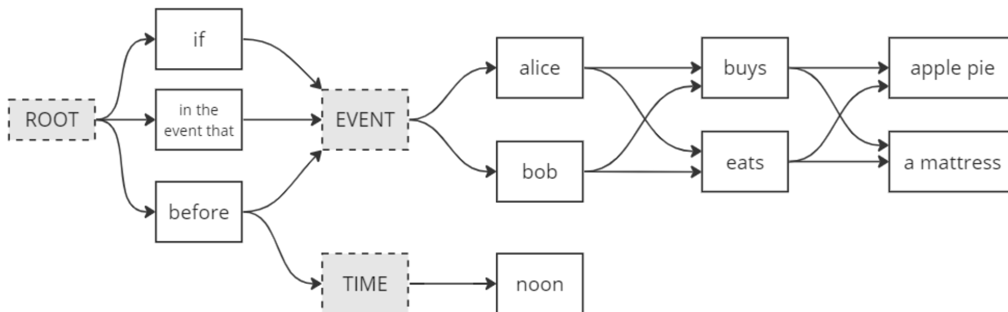


Figure 5.3: Simple CNL selection tree

We will refer to the components that make up a pattern class as *pattern variables* (e.g., P_BEFORE_X, EVENT_X). To construct this tree from our grammar, we will need to iterate through the three pattern classes, identify all forms that each pattern variable can take, and configure the proper parent-child relationships. For example, to find the children of the root node, we need to find all forms that the pattern variables P_BEFORE_X and P_CONDITIONAL_X can take, since those are the first pattern variables among all pattern classes. The specific values that each of these pattern variables can take become the concrete *input units* that can be selected by a user. This yields the values ‘if’, ‘in the event that’, and ‘before’. For each of these nodes, we need to look at the next possible values in the grammar, and add those as further children. The technical details on the specification of our final CNL in SymboleoNLP and the recursive algorithm for the construction process of the tree that represents the grammar can be found in Appendix C.

Once the tree is constructed, it is conceptually straightforward to implement the selection of a valid pattern class. Once the user has chosen a parameter to refine, they will be presented with the three children of the root node. The user then selects one of these nodes, the value gets stored, and we present the children of this selected node. This continues until a leaf node is reached, at which point we are guaranteed to have a valid pattern class. This approach would solve both sub-tasks, thus completing the requirement. This illustrates a general solution to implementing an EBNF grammar in the context of our tool. Our CNL is much more complex than this simple example, but we can use the same general principle. There are a number of libraries that can work with EBNF grammars that may have potential value in future work. The *pyparsing* library [2], for example, is able to represent and verify sentences against specified EBNF grammars. However, it does not have an easy way of iteratively collecting input with the tree traversal method discussed, which is part of our first requirement.

5.2.2 Range of Input

The first complexity worth mentioning is that our CNL is much *bigger* than this simplified one. We have many more pattern classes, and the pattern variables that make up these pattern classes can take a variety of different input units. We use the same construction and tree-traversal approaches, but we acknowledge that the tree will be much larger. The bigger issue, to which we will be devoting the majority of this discussion, is that our grammar is *dynamic*. Some of the pattern class components are not fixed and will require user input. This dynamism was included in the CNL since it adds a degree of expressiveness found in real contracts, but now we must face the drawback of this added complexity. Our simplified CNL restricted the verbs that could be used to two simple verbs. We make no such restriction on our full CNL, since we want to allow the user flexibility to specify many types of events that we cannot predict in advance. This is also the case with many other event components (subject, predicate, adverb, etc.) as well as other temporal components (date, timespan units/values, time period, etc.).

This fact means that, in some cases, the user can enter *freeform text* (assuming some sort of form-based input mechanism on the UI). This has the potential to introduce ungrammatical input, which is not a problem we faced with our simplified CNL. The user can purposefully enter complete nonsense text, or they may *accidentally* enter ungrammatical text. For example, instead of entering the refinement `'within 2 weeks of the renter occupying the property'`, they might make an error on the verb tense and enter `'within 2 weeks of the renter occupies the property'`, which is grammatically *incorrect*. Furthermore, this also allows for *semantic* incorrectness, for example if the user enters `'within 2 weeks of the renter driving the waterfall'`. Note however that semantic incorrectness was possible even with our simplified CNL (e.g., `'if Alice eats a mattress'`).

Certain textual components *are* fully under our control in this tree-traversal process. For example, some of our input units correspond to simple words or phrases ('if', 'within', 'in case'). We call these input units *static*. Static units have fixed values and do not

[2] `https://pypi.org/project/pyparsing/`

require any user input. Dynamic values require user input and are therefore a source of error. A third variety of unit in addition to static and dynamic is the *empty* unit. An empty unit is simply used to indicate the type of entity that follows. The EVENT unit is an example; It does not correspond to any text, but it allows the user to explicitly state that they want to enter an event, rather than a date, for example. This is also seen in our simplified CNL tree in Fig. 5.3.

To further make these concepts more explicit for the sake of the user, we have two specific *types* of events.

- A *contract event* simply refers to an event that happens to the contract itself, for example contract activation or termination.
- A *custom event* is any dynamic event that might be taken by one of the contracting parties. This is the type of event that we are familiar with from our discussion on our *standard event* specification.

Each type of event is also assigned as an empty unit. The *custom event* is the more complex type and will be the main focus of our attention. All custom events must begin with a subject, so the Subject unit is the only child of the empty CustomEvent unit. An illustrative sub-tree of our full CNL is presented in Fig. 5.4.

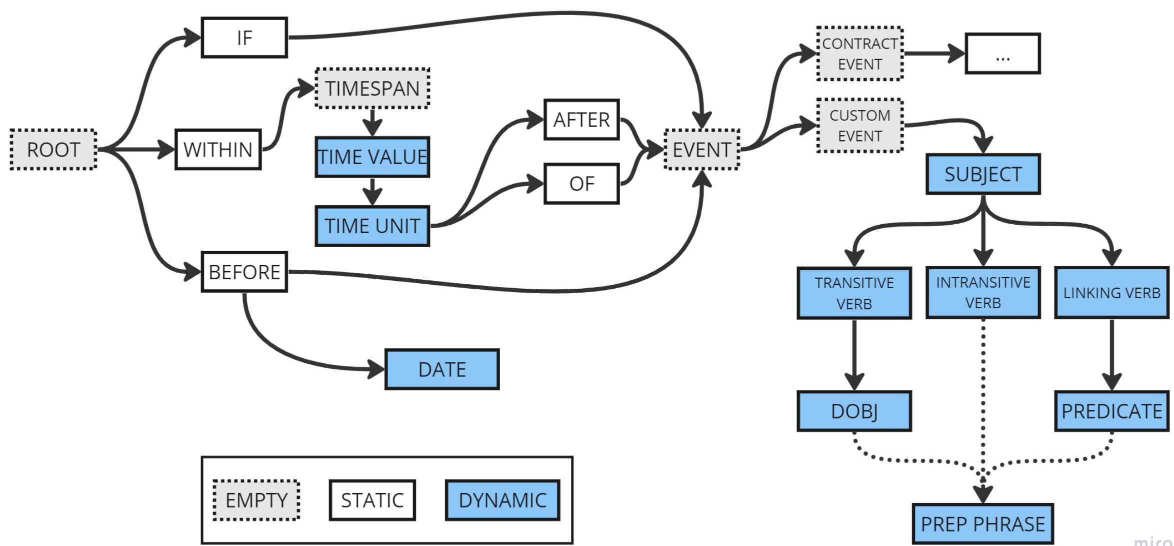


Figure 5.4: Subset of the input unit selection tree

To generate an input, the user must traverse the tree by selecting nodes, each of which corresponds to an input unit type. For each non-empty input unit, a textual value is required. If the unit is static, it will automatically use the static string value associated with the node (e.g.,'if', 'before'). If the unit type is dynamic, the user will be prompted to enter a string value for it. Once the value for a node is obtained, the tree traversal continues, and the user is presented with the children of the previously selected node. This process continues until we either reach a leaf *or* a *final node* is encountered. A *final node* is useful in places where the input is *optional*, for example on the PREP_PHRASE node.

5.2.3 Mitigating User Errors

We want to manage our expectations for what a tool like SymboleoNLP should provide in attempting to avoid errors associated with freeform user input on the dynamic input. The types of errors introduced by our dynamic units are errors that would be possible with *any* application that allows expressive user input. It would be possible to integrate some sort of grammar checker into our application to help mitigate this type of error. Grammar-checking mechanisms on many existing applications (such as word processors) are capable of detecting both syntax and semantic errors. Our application is built using object-oriented design as well as SOLID design principles [61], which facilitates easy integration of new modules. We therefore leave the possibility of integrating syntax and semantic-checking modules to future work, but we note that these modules will necessarily be probabilistic, and therefore cannot guarantee the identification of syntactic or semantic errors.

We will, however, implement *one* useful step towards facilitating valid user input. Unlike general purpose word processing applications, we are constrained to specific domains. We are operating in the *legal* domain, so it is likely that the user will be entering text commonly seen in the legal domain. Furthermore, we are working with a specific contract whose domain is specified in the *domain model* of the contract itself. Therefore, it is also likely that the user will be refining the contracts using concepts *already found* in the domain model. In our example, the subject is the 'renter', which will correspond to an existing domain model role. Therefore, for some of these dynamic components, we can integrate suggestions based on the domain model and the legal domain. For example, when the user is prompted to enter the direct object for an event, we can provide a list of roles and assets from the domain model. The user is not restricted to *only* these values, but we can at least provide them as options. Since this is largely a UI concern, we will not discuss this in more detail. We will, however, note that this is an example of a more general approach of inferring some properties about future customizations based on existing contract. In future work, there may be the possibility of using pre-existing knowledge and also initial customizations to make intelligent inferences about further refinements that might be done to the contract. An example may be that we can infer the time unit of a customization (e.g., days, months, weeks) based on a previous customization where the contract author entered that time unit.

In summary, the concept of dynamic units in our CNL opens up the possibility for user errors. These may be mitigated by introducing syntax/semantic verification modules to the input process, or by integrating domain-based suggestions from the legal domain or the domain model itself. While this dynamism is a source of error, it does not affect our fundamental tree-traversal selection strategy for generating valid inputs.

5.2.4 Assumption of Linguistic Knowledge

While our tree-traversal approach guarantees that the input unit *types* correspond to our CNL, it makes an assumption that the user is familiar with some basic linguistic terminology. For example, once the user indicates that they want to enter an event, they will be presented with a `SUBJECT` unit. Once they enter a subject, they will then

be presented with three options: a TRANSITIVE_VERB, an INTRANSITIVE_VERB, and a LINKING_VERB. These verb types are used for different purposes, and thus will have different child nodes:

- **Linking:** A linking verb is followed by a predicate, which can take the form of an adjective ('legal proceedings become necessary').
- **Transitive:** A transitive verb is followed by a direct object, which can take the form of a noun phrase ('The contractor completes the services').
- **Intransitive:** An intransitive verb is not necessarily followed by anything ('The contract terminates').

Verbs can also be divided into *action* verbs (e.g., pay, deliver, disclose) and *stative* verbs (e.g., know, is, has). Our grammar does not formally distinguish between these two, although there is some overlap between linking verbs and stative verbs. Linking verbs are typically used to *describe* a subject (the flower *is* red) rather than state an event occurrence, however the verb 'become(s)' is an interesting case. While it is considered a linking verb, it can also semantically be considered an event. We support this use-case of become(s) as a linking verb and treat this as an event, similar to the transitive and intransitive verb phrases. Future work may include mapping stative events to more specific SYMBOLEO constructs.

Since the user will be presented with the unit types through the tree-traversal generation approach, it is assumed that the user will understand the meaning of concepts such as transitive/intransitive/linking verbs. The target user for a tool like this would be a contract author, so this *may* be a valid assumption, but it is an assumption nonetheless. We noted that a key motivation for this tool was to ensure that contract authors do not need to understand a specification language like SYMBOLEO. A trade-off is that it *does* require that they understand these linguistic concepts. If the event specification is made more complex, then the number of linguistic concepts may increase (e.g., subordinating conjunction, verb polarity, pronoun resolution, etc.). This may be further exacerbated by the fact mentioned earlier that some linguistic concepts (such as the preposition) are not universally agreed upon.

A potential mitigation strategy could be to offer some clarifying examples and explanations of these concepts in the UI. An even more sophisticated system may integrate further syntax and semantic verification modules into the flow that could *infer* the unit types. For example, rather than having the user explicitly select the verb type (transitive/intransitive/linking), they could simply enter *any* text, and a module would confirm whether or not it is a verb, figure out which *type* it is, and figure out if it is a stative or action verb as well. While the incorporation of such a module is indeed possible, we would caution that the mechanisms will be largely probabilistic and thus prone to error. It is far from trivial to automatically determine the type of a given verb. Some verbs are easy to classify, but for many others, their classification is context-dependent. For example, the verb 'appear' can be linking or intransitive, and the verb 'terminate' can be transitive or intransitive. We therefore acknowledge this assumption of linguistic knowledge as a limitation, but note that there are non-invasive (though probabilistic) methods for removing the need for this assumption as potential future work.

Listing 5.5: CNL input generation algorithm

```python
def generate_cnl_input(contract: SymboleoContract, tree_root: GrammarNode) -> List[
    UserInput]:
    unit = tree_root
    results: List[UserInput] = []

    while True:
        children = get_unit_children(unit, contract)

        if len(children) == 0:
            break
        else:
            # User will select one of the child nodes
            unit = manual_select_child(children)

        if unit.variety == 'STATIC':
            input_value = get_default_value(unit)
        elif unit.variety == 'DYNAMIC':
            input_value = get_user_value(unit)
        elif unit.variety == 'EMPTY':
            input_value = get_empty_value(unit)

        results.append(input_value)

    return results
```

5.2.5 Sample Refinement

Now that we have discussed the sources of error in our approach, we present a simplified form of the algorithm for generating valid CNL patterns in Listing 5.5. The inputs to this function include the Symboleo contract that we are customizing as well as the root node of the tree representing our grammar. A representation of the results of this process is shown in Table 5.3.

The get_unit_children function will fetch the children of the most recently selected node on the tree (or the children of the root on the first iteration). We also pass the contract into this function so that we can populate the children with our 'suggestions' from the domain model and declarations. The manual_select_child function involves the user selecting one of the generated children, and the various value-getting functions will either collect user input (if the unit is dynamic), return the static value (if the unit is static), or return an empty unit (if the unit is empty).

We will walk through this algorithm by showing how the refinement '`within 2 weeks of renter occupying the property`' would get generated. A series of screenshots illustrate this functionality as implemented by SymboleoNLP.

1. The user selects pay_deposit.P1 (Fig. 5.5), and the children of the root node are presented to the user (Fig. 5.6). These include 'If', 'In the event that', 'Before', 'Within', etc.

2. The user selects the static unit 'Within' from this set of children. Since this is a static unit, the value for the input unit is simply 'within'. This is added to our set of results as our first value, and the loop continues.

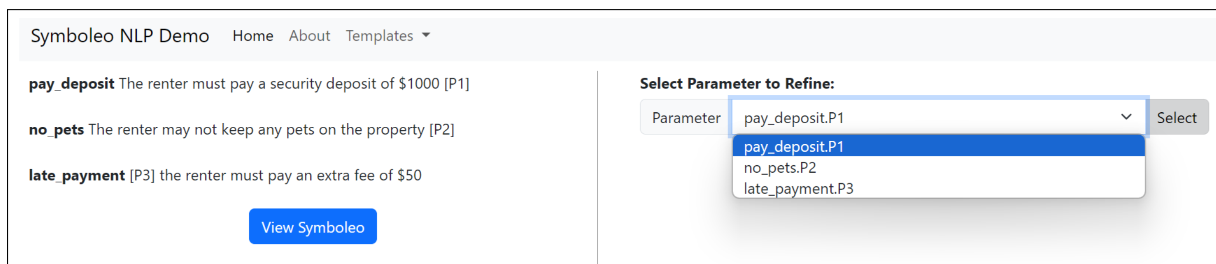


Figure 5.5: User selects a parameter to refine

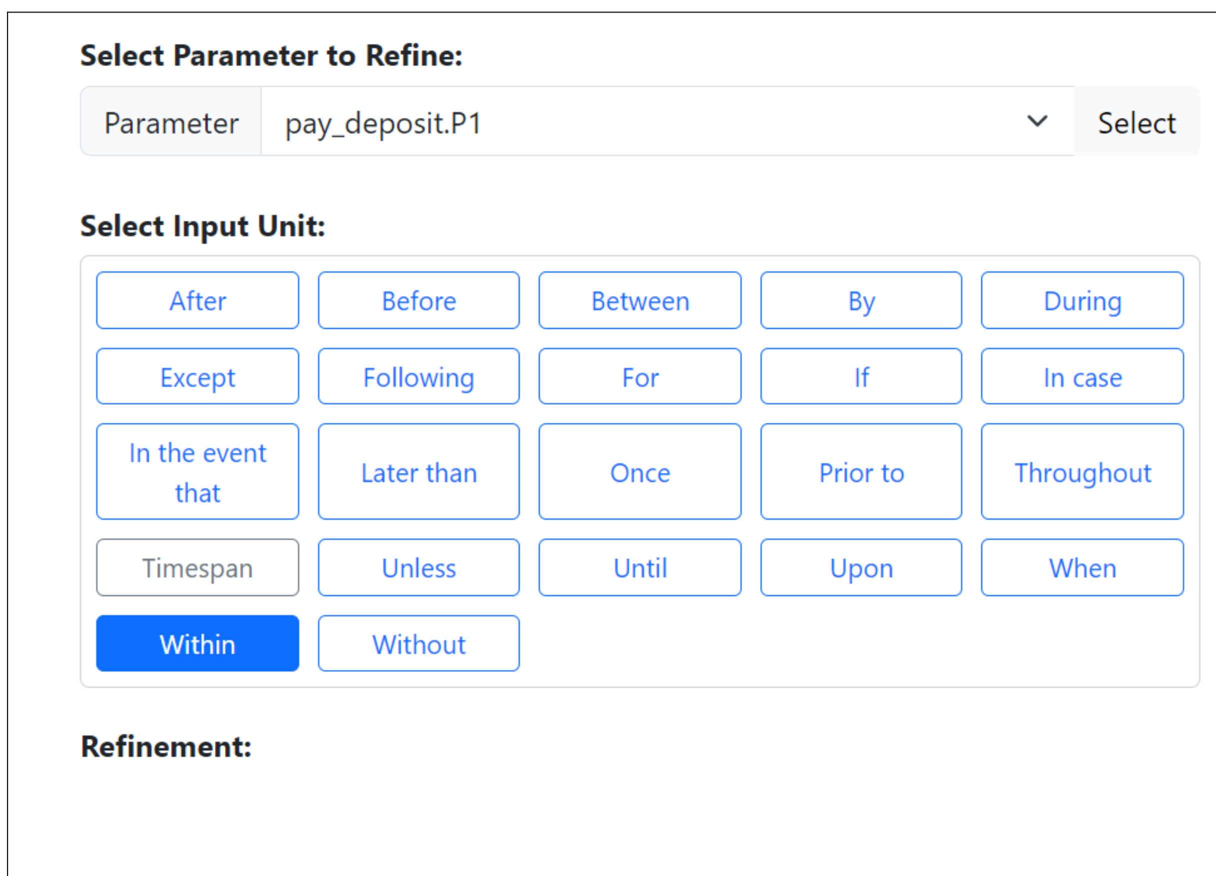


Figure 5.6: User is presented with all children of the root node

3. The children of WITHIN are fetched, one of which is the TIMESPAN unit. This is selected by the user.

4. The TIMESPAN unit is an empty unit, used for the explicit indication that the user is entering a timespan. The timespan is followed by two dynamic units, the TIME_VALUE (a number), and the TIME_UNIT (days, weeks, months, ...). Both of these are dynamic.

5. The user is first prompted for a value for TIME_VALUE and enters '2'. They are next prompted for a TIME_UNIT and they enter 'weeks' (Fig. 5.7). Both of these are added to our result and the loop continues. Note that this is a potential source of user error, since the user could enter anything into the dynamic values.

Select Parameter to Refine:

Parameter | pay_deposit.P1 | Select

Select Input Unit:

Time unit

Select Value:

Time unit | weeks | Select

Enter Custom:

Time unit | | Add Custom Value

Refinement:

`within 2`

Figure 5.7: User selects the 'weeks' option for the time unit of the timespan component

6. The next children on the tree are fetched. These will be all of the members of the P_AFTER_W class ('following', 'of', 'after').

7. The user selects OF, which is another static node. The value 'of' is added and the loop continues.

8. The empty Event node is the only child, so it is selected. The user is now presented with the different event types (CustomEvent, ContractEvent). The user selects CustomEvent, another empty unit, indicating that they want to enter a custom event.

9. The user must now enter the custom event according to the grammar, starting with the subject, which is the only child of the custom event unit. The user is presented with the values of assets and roles from the domain model as optional suggestions. They choose the 'renter' role, and the value of 'renter' is added to our input.

10. The children of the subject node are all three verb types (intransitive, linking, transitive). This is where we rely on the assumption that the user has sufficient linguistic knowledge and selects the Transitive Verb (Fig. 5.8). As a value for this dynamic unit, they enter 'occupying' (Fig. 5.9).

11. The direct object node follows from the transitive verb, and the user enters the value of 'property'. Once again, the property will already be a declared asset from the domain model, and is therefore included in the list of suggestions presented to the user.

12. At this point, the only child will be a prepositional phrase node, which is optional. NL technically allows for infinite prepositional phrases (e.g., the bump on the log at

Figure 5.8: User is presented with 3 types of verbs

Figure 5.9: User enters a dynamic value for the transitive verb

the bottom of the sea...), but our CNL is currently restricted to one. The input is now complete (Fig. 5.10), and the user can submit the refinement.

Select Parameter to Refine:

Parameter pay_deposit.P1 Select

Select Input Unit:

Adverb Preposition

Refinement:

`within 2 weeks of renter occupying property`

Submit

Figure 5.10: Completed user entry

5.2.6 Evaluation

The bulk of this section has focused on our first sub-task of ensuring the user can *only* enter valid CNL patterns. We discussed a tree-traversal algorithm and noted the potential sources of error with this approach, arising from the fact that we want to allow for dynamic user input, which is crucial to capturing the expressiveness of our CNL. The tree-traversal selection can at least guarantee that the input unit *types* will correspond to one of our pattern classes. As long as we have included our entire set of pattern classes in our grammar, then this also addresses our *second* sub-task.

Since we are now working with application code, it is important to further evaluate all functionality with a test suite consisting of unit tests and integration tests. Unit tests are written against each individual function to confirm that the function operates as expected. This includes testing all *branches* within a function. We used an object-oriented approach to building this tool, keeping functions focused on single tasks, and making use of dependency injection for easy testing. Through carefully designed unit testing, we are able to achieve 100% code coverage on this set of functionality (Fig. 5.11), with all tests successfully passing. We temper this achievement however, by reminding ourselves that testing can only prove the *presence* of bugs, not the *absence* [23]. To help further mitigate issues in our code, we can also add integration test suites, which test the full operation of our key processes.

The primary test suite for this requirement is structured as follows. We create a sequence of unit types for each pattern class in our CNL. For each sequence, we iterate

	Input Unit	Variety	Value	Children
1	ROOT	empty		WITHIN, IF, BEFORE...
2	WITHIN	static	within	TIMESPAN
3	TIMESPAN	empty		TIME_VALUE
4	TIME_VALUE	dynamic	2	TIME_UNIT
5	TIME_UNIT	dynamic	weeks	OF, AFTER, ...
6	OF	static	of	EVENT
7	EVENT	empty		CUSTOM_EVT, CONTRACT_EVT
8	CUSTOM EVENT	empty		SUBJECT
9	SUBJECT	dynamic	renter	T_VERB, L_VERB, I_VERB
10	T_VERB	dynamic	occupying	DOBJECT
11	DOBJECT	dynamic	property	PREP_PHRASE

Table 5.3: Input units for P1 refinement

Coverage report: 100%

coverage.py v6.4.4, created at 2023-08-26 19:44 -0400

filter...

Module	statements	missing	excluded	branches	partial	coverage ↑
app\classes\events\contract_events.py	13	0	0	12	0	100%
app\classes\events\custom_event\adverb.py	21	0	0	6	0	100%
app\classes\events\custom_event\custom_event.py	48	0	0	22	0	100%
app\classes\events\custom_event\noun_phrase.py	32	0	10	12	0	100%
app\classes\events\custom_event\predicate.py	6	0	0	2	0	100%
app\classes\events\custom_event\prep_phrase.py	9	0	0	2	0	100%
app\src\pattern_builder\pattern_unit_fillers\contract_action_filler.py	21	0	0	2	0	100%
app\src\pattern_builder\pattern_unit_fillers\custom_event_filler.py	17	0	0	2	0	100%
app\src\pattern_builder\pattern_unit_fillers\pattern_unit_filler.py	11	0	1	4	0	100%
app\src\pattern_builder\recursive_pattern_checker.py	33	0	1	24	0	100%
app\src\pattern_builder\single_pattern_checker.py	30	0	1	14	0	100%
Total	2912	0	123	967	0	100%

Figure 5.11: Screenshot showing 100% code coverage on SymboleoNLP

through the tree, ensuring that the sequence can be found by traversing the tree. This guarantees that every pattern class is represented. We also make use of a PatternExtractor class in our code, which verifies that the list of input types exists as a potential form of a pattern class in the CNL. In this way, the PatternExtractor class acts as the reverse of the tree construction algorithm – it recursively checks the CNL grammar to confirm that the test value matches the predicted pattern class. There is a test value for each pattern class. The pseudocode for the PatternExtractor can be found in Appendix D. An illustrative example of a pattern class test is found in Listing 5.6.

To evaluate our second sub-task of ensuring that *only* input types that correspond to our CNL are entered, we construct a test function that traverses the tree and construct a list containing all possible paths. Each path in this list is then tested against the pattern class extractor to ensure that it matches with a valid pattern class. Now that we have a clearer understanding of how valid CNL is generated by our tool, we can focus on the second key requirement of mapping this CNL to the correct Symboleo refinement operations.

Listing 5.6: Sample pattern class test

```
1  def test_pattern_class():
2      test_value = [
3          UnitType.WITHIN,
4          UnitType.TIMESPAN,
5          UnitType.TIME_VALUE,
6          UnitType.TIME_UNIT,
7          UnitType.OF,
8          UnitType.EVENT,
9          UnitType.CONTRACT_EVENT,
10         UnitType.CONTRACT_SUBJECT,
11         UnitType.CONTRACT_ACTION
12     ]
13     expected_class = WithinTimespanEvent
14
15     pattern_classes = get_all_pattern_classes()
16     curr_node = construct_tree(pattern_classes)
17     contract = get_test_contract()
18     i = 0
19
20     # Traverse the tree, following the path prescribed by the test_value
21     while i < len(test_Value):
22         input_list = get_node_children(curr_node, contract)
23         next_input = [x for x in input_list if x.unit_type == test_value[i]][0]
24         curr_node = [x for x in curr_node.children if x.name == next_input.unit_type][0]
25         i += 1
26
27     # Now verify that the test_value matches the expected pattern class
28     result = extract_pattern_classes(test_value)
29     assert(expected_class in result)
```

5.3 Mapping CNL to Symboleo

5.3.1 Overview and Algorithm

In addressing the mapping requirement, we assume that we are working with a valid and grammatical CNL refinement, represented by a list of unit types and their values, as described in the previous section. We emphasize that a refinement does not necessarily correspond to a *single* SYMBOLEO operation. If the refinement introduces a new event or asset, then those new artifacts must be added to the domain model and as declarations. We therefore include a *norm update extraction* component and a *domain update extraction* component for each refinement. The former is responsible for refining the target norm (e.g., mapping the consequent from a `Happens` predicate to a `WhappensBefore` predicate and adding the required additional argument). The latter is responsible for identifying and extracting new events and assets as domain classes as well as their corresponding declarations. The result of these two components will be an *update object*, which will contain a set of SYMBOLEO entities that can then be applied to the existing SYMBOLEO contract. We will continue with our example of refining the first parameter of our sample rental contract, the input of which is found in Table 5.3. The goal of this step will be to extract the following three target operations, which will recreate the refined obligation from the original contract:

1. Refine the `Happens` predicate on the consequent to `WhappensBefore` with the proper arguments:

- WhappensBefore(evt_pay_deposit, Date.add(evt_occupy_property, 2, weeks))

2. Add the domain event class of occupying the property:

   - OccupyProperty isAn Event with property: Property, occupier: Role;

3. Add a declaration for the occupying event:

   - evt_occupy_property: OccupyProperty with property := the_property, occupier := renter

5.3.2 Pattern and Event Extraction

A pre-requisite to our extraction components is that we use our recursive PatternExtractor functionality, which was introduced for testing purposes in the previous section. We use this functionality to extract a PatternClass entity from our list of input units, which will be used directly by our extraction components. In cases where the result of the pattern extraction is *multiple* pattern classes, we require an additional pattern-resolution step to resolve this list into a single pattern class. Furthermore, if the pattern class contains an EVENT pattern variable, then we are going to need access to certain linguistic properties of the event, which are not yet present in our basic list of input values. We therefore need to extract a more complex event entity, called the CustomEvent, from our input list. These linguistic properties will be used to construct standardized formats for the event reference names and the domain model attributes. Some of the linguistic properties that we want to extract include the verb lemma, noun phrase heads, and asset types of noun phrases. We therefore create an EventExtraction function, which will parse through our input list and extract all of this linguistic information into a CustomEvent object, which will be made accessible in the domain update and norm update extraction steps. Since our input into this extraction step is simply a set of text values, we will need to use a set of NLP techniques to extract this more sophisticated information.

As an example, consider the event '...dolphin receiving the digital photo files'. The only linguistic information we have about this event is that certain word groups are associated with unit types. For example, 'dolphin' is the SUBJECT, 'receiving' is the TRANSITIVE_VERB (T_VERB), and 'the original photo files' is the DIRECT_OBJECT (DOBJ). The Python specification for the extracted CustomEvent can be found in Listing 5.7. We will see how these linguistic properties are used in the subsequent steps. Further details of this extraction can be found in Appendix E.

Listing 5.7: CustomEvent sample specification

```
1 dolphin = NounPhrase(
2     str_val = 'dolphin',
3     head = 'dolphin',
4     is_plural = False,
5     is_role = True,
6     determiner = None,
7     adjectives = [],
8     asset_type = 'Role'
```

```
)

photo_files = NounPhrase(
    str_val = 'the original digital photo files',
    head = 'files',
    is_plural = True,
    is_role = False,
    determiner = 'the',
    adjectives = ['original', 'digital', 'photo'],
    asset_type = 'Files'
)

verb_receive = Verb(
    verb_str = 'receiving',
    lemma = 'receive',
    verb_type = VerbType.TRANSITIVE,
    conjugations = VerbConjugations(
        present_singular = 'receive',
        present_plural = 'receives',
        past = 'received',
        continuous = 'receiving'
    )
)

full_event = CustomEvent(
    subject = dolphin,
    verb = verb_receive,
    dobj = photo_files
)
```

5.3.3 Norm Update Extraction

With the pattern extracted and a more sophisticated event specification (for event-based pattern classes), we can now address the two main components of the operation mapping. The norm update extraction component is primarily informed by the mapping defined by our CNL. The pattern class within TIMESPAN P_AFTER_W EVENT corresponds to a `Happens` → `WhappensBefore` operation. For each pattern class, we define a handler function to handle the input. For example, we define a WithinTimespanHandler class, which is called when the pattern class represented by the input is within TIMESPAN P_AFTER_W EVENT.

This handler will create a new `WhappensBefore` predicate, which requires two arguments. The first is the base event (event X in 'X happens before Y'). In our example, this refers to the event of paying the deposit. This was the initial event in our unrefined `Happens` predicate, so we simply copy that event reference. The second argument, the point function, is new, and must be extracted from our pattern class. A point function specifies some period of time relative to an anchor event. It has three arguments: The first is the anchor event ('the renter occupying the property'), the second is the time value (2), and the third is the time unit (weeks). One extra consideration for the time value is that we need to determine if the time is relative to *before or after* the anchor point. The meaning of this pattern class is that it is 2 weeks *after* the event. If it were *before*, we would need a *negative* value (-2) for this argument, indicating that it is 2 weeks *before* the anchor event.

By virtue of our previous pattern-extraction step, these values are easily accessible on the pattern class object and we use them to fill the required arguments. Note that in a Symboleo obligation, this event manifests as a reference to an *event declaration*

(e.g., `Happens(evt_occupy_property)`). This event, however, does *not yet exist* as a declaration, as it will be taken care of in the domain update extractor. We generate the reference name (evt_occupy_property) using the components from our CustomEvent component that we extracted in the previous step. For transitive verb phrases, we simply concatenate the verb lemma (occupy) with the head of the direct object noun phrase (property), and pre-pend with 'evt' to get evt_occupy_property. Intransitive and linking verbs have slightly different rules for generating event reference names, but they all make use of the building blocks extracted during the event extraction using NLP techniques. This results in a more standardized format of our event references.

We now have all the components of our new predicate. We create a new obligation using this updated predicate. This predicate applies to the consequent, and all other components of the initial obligation remain intact (debtor, creditor, trigger, antecedent). This updated obligation is returned from the norm update extractor. We present the simplified code for the WithinTimespanHandler in Listing 5.8, and note that each handler will have code specific to the required refinement operation:

Listing 5.8: Norm update extractor for the sample pattern class

```
class WithinTimespanHandler:
    def handle(pattern_class: WithinTimespanEvent, norm: Norm) -> Norm:
        # Get the initial event from the unrefined norm
        initial_event = norm.get_default_event('consequent')

        # Extract entities from the pattern class
        evt = pattern_class['EVENT']
        time_value = pattern_class['TIME_VALUE']
        time_unit = pattern_class['TIME_UNIT']

        # Create the point function using our extracted entities
        point_function = PointFunction(evt, time_value, time_unit)

        # Create the new predicate and update the norm
        new_predicate = WhappensBefore(initial_event, point_function)
        norm.update('consequent', new_predicate)

        return new_norm
```

5.3.4 Domain Update Extraction

The extraction of relevant domain updates and declarations can be considered mostly independent from the extraction of the norm update. This component is only required if a new event is introduced by the CNL refinement. In our example, we have introduced the event of the renter occupying the property. At the very least, we will need to add a domain class to the domain model that represents this event as well as its corresponding declaration. In some cases, we would also need to add any new assets that are referenced in the new event. In our case, the only potential asset is the 'property', which is already included on the broadened contract template. We therefore only need to focus on the domain event and corresponding declaration.

We must make a few important notes to preface this discussion. First, we have emphasized that we must extract both a domain class *and* a declaration. We will be focusing mainly on the extraction of the more difficult piece, the declaration. The

declaration is an *instance* of a domain class, so in a sense we are working backwards. However, if we can identify the declaration, then the domain class is trivially deduced from the declaration. Therefore this discussion will focus on extracting declarations. This extraction will rely on some important assumptions about the linguistic properties of our events, which will be another potential source of error. We will highlight these assumptions and the corresponding threats they present, and ideate on potential future work for mitigating these threats.

An event declaration represents an event as a set of key-value properties. The keys (property, occupier) refer to important elements in specifying the meaning of the event. The event of 'occupying' implies that there is an agent that is doing the occupying (the occupier) and a location that is being occupied (the location). We can therefore conceive of an event as a pre-defined pattern with a set of slots to be filled, some of which may be optional or mandatory. This is reminiscent of frame semantics [4], where a *frame* has a list of *frame elements* that define the event. This presents us with two related challenges:

1. **Identifying the slots:** Given an event specified in our CNL (e.g., 'renter occupying the property'), how can we automatically determine what the the required frame elements/slots are (e.g., 'occupier' and 'location')?

2. **Filling the slots:** Supposing we have identified a set of required slots, how do we automatically populate it with the components of our CNL event? For example, how can we know that we should put 'renter' into the 'occupier' slot, and 'the property' into the 'location' slot?

These are non-trivial problems to solve, and we must once again face the inherent variety of NL and the complexities it creates. Any solution to these problems will necessarily be rooted in heuristics and/or probability, and thus prone to error. We will now discuss some of the basic heuristics we apply to these challenges.

5.3.5 Identifying and Filling Event slots

Given the statement of an event in our CNL, we need to identify the slots that give *meaning* to the event. To provide a sense of the scope of this challenge, we present some simplified examples from our real contract dataset that match our event specification:

- ...Dolphin receiving the original digital photo files
- ...Bosch disrupts productivity of Client
- ...Buyer opens an irrevocable letter of credit

In our first example, our verb is 'receiving'. It contains a 'receiver' (Dolphin) and a received object (original digital photo files). The second example (disrupts) has a disruptor (Bosch), the thing that is disrupted (productivity), and the target of the disruption (Client). The third example (opens) has an opener (Buyer), the thing that is opened (irrevocable letter), and a further descriptor for the letter (credit). These examples highlight some

useful heuristics. For a given verb, we have an *agent* that is performing the event. This typically takes the role of the subject in the event specification. We can construct this slot name by appending the word 'agent' to the continuous verb form (occupying_agent = renter, receiving_agent = Dolphin, disrupting_agent = Bosch, opening_agent = Buyer).

The events also typically have an object that the action is being performed on, which typically refers to the direct object in a transitive verb. In many cases this might be an asset (letter, property), in which case we can generalize the slot to the past-tense form of the verb followed by the word 'object' (occupied_object = the property, received_object = original digital photo files, disrupted_object = productivity, opened_object = irrevocable letter). If the direct object is a *role*, then it may be more appropriate to use the word *target* instead of *object*. We have access to these conjugated forms of the verb, as well as other useful linguistic properties, via our `CustomEvent` object that we extracted earlier. Our event specification also allows for prepositional phrases, which can correspond to additional types of information associated with the event. We can handle this differently depending on the information that is available to us from the CNL event. For example, if the preposition used is 'with', and the prepositional object is a role, then perhaps we can label the slot as a 'co_agent'. If the prepositional object is *not* a role, then perhaps the slot is labeled as a 'method' (e.g., '*with* a credit card').

With these heuristics in mind, we make the following mappings on our CNL event:

1. The subject (renter) maps to the occupying_agent slot
2. The direct object (property) maps to the occupied_object slot

The event name for our declaration is the same one we used for the event reference (evt_occupy_property) in the norm update extraction step. Depending on the verb type, we concatenate the verb lemma with other components of the event to come up with this intuitive name. We use the same principle for coming up with the event *type*, which is 'OccupyProperty' in this case. We have now created our declaration: `evt_occupy_property: OccupyProperty with occupying_agent := renter, occupied_object := the_property;`.

From this declaration we can trivially deduce the corresponding domain event:
```
OccupyProperty isAn Event with occupying_agent: Role,
occupied_object: Property;
```
.

To clarify, 'OccupyProperty' is a domain event, meaning it is a class from which specific events can be defined. Any instance of 'OccupyProperty' will require an 'occupying_agent' (filled by a role) and an 'occupied_object' (filled by an asset with the type 'Property'). The declaration 'evt_occupy_property' is such an instance of this domain event. The result of our domain update extraction is these two artifacts. Note that these are not exactly identical to our initial target operations (we have occupying_agent instead of occupier). In this case, the meaning is synonymous, highlighting the fact that this step leaves room for such flexibility.

In general, our approach to the declaration extraction problem involves using rough heuristics to map the noun phrases (subject, direct object, prepositional object) present

in an event into inferred slots. While some of the heuristics are based on observed regularities in the contract texts we analyzed, this is *not* a rigorous process, and is thus prone to errors. Furthermore, this process can potentially become even more difficult as the event specification becomes more sophisticated. In fact, this challenge may represent a limiting factor in deciding whether or not to extend the event specification. If events cannot be reliably mapped to declarations, then we may not want to attempt to represent more complex events in the grammar. There is potential for future work to take a more evidence-based approach to this, which would involve closely analyzing contract events, perhaps by using the same dataset we have introduced in this work. There may also be room for further integrations with modules that can help with this. There is Framenet functionality within NLTK [3], which allows one to find frames using keywords and obtain a list of the frame elements. Advanced probabilistic methods based on transformers or neural networks may also have some potential, but this would require a labelled dataset to learn on or a more sophisticated mechanism to transfer a trained model from another domain.

5.3.6 Completing the Mapping

We have now identified all of the required artifacts. We have a refined norm, a declaration, and a domain class. The final step is to apply these entities to the existing SYMBOLEO specification. The new norm will replace the existing norm, and the domain class and declaration will be newly added. We now have an updated SYMBOLEO contract. By our principle of integrity, this contract is still syntactically correct. We present the simplified algorithm for this mapping process in Listing 5.9. This flow is activated once the user submits their CNL refinement. The NL template is updated accordingly with the user's input (Fig 5.12). The web tool for SYMBOLEONLP also allows the user to view the generated SYMBOLEO code, and the updated code is shown in Fig 5.13. Another component of a SYMBOLEO contract that we have not not addressed in-depth is the contract specification parameters (CSPs). These should not be confused with the contract template parameters (e.g., [P1], [DELIVERY_REFINEMENT]). CSPs refer to the parameters that must be filled with arguments in the contract signature. These are also addressed by the tool, and a more in-depth discussion their handling can be found in Appendix F.

Listing 5.9: Operation mapping algorithm

```
1  def map_cnl_to_operations(contract: SymboleoContract, initial_norm: Norm, input_list:
       List[InputUnit]) -> SymboleoContract:
2      # Extract the pattern classes
3      pattern_classes = extract_pattern_class(input_list)
4
5      # If there are multiple possible pattern classes, then we must resolve them to one
6      pattern_class = resolve_pattern_class(pattern_classes, initial_norm, contract)
7
8      # Extract the CustomEvent for Event-based pattern classes
9      if pattern_class isinstance(EventPatternClass):
10         custom_event = extract_event(pattern_class)
11         pattern_class.event = custom_event
12
13     # Fetch the required handler based on the pattern_class
```

[3] https://www.nltk.org/howto/framenet.html

```
14        norm_handler = HANDLER_DICT[type(pattern_class)]
15
16        # Norm update extraction
17        norm_update = norm_handler.handle(pattern_class, initial_norm)
18
19        # Domain update extraction
20        domain_updates = extract_domain_updates(pattern_class, contract)
21
22        update_object = ContractUpdateObject(
23            domain_updates,
24            norm_update
25        )
26
27        contract.apply_updates(update_object)
28
29        return contract
```

pay_deposit The renter must pay a security deposit of $1000 within 2 weeks of renter occupying property

no_pets The renter may not keep any pets on the property [P2]

late_payment [P3] the renter must pay an extra fee of $50

View Symboleo

Figure 5.12: Updated NL template

Much of this section has been spent discussing the domain update extraction, which we acknowledge is of secondary importance to the norm update extraction. The discussion for the norm update extraction was much simpler, largely due to the fact that our CNL construction focused on coming up with these mappings for the norm updates, where we took an evidence-based approach informed by linguistic heuristics.

The mappings provide a clear set of requirements for SymboleoNLP. Mappings for all of the *refinement*-based pattern classes are covered. This excludes the pattern classes that correspond to the addition of a new contract termination power. The COND_A EVENT and COND_T EVENT pattern classes can map to either a refinement (i.e., adding an antecedent or trigger to an existing norm), or they can map to the creation of a contract termination power, as discussed in Sec. 4.4. The tool currently supports the *refinement* mappings. The P_EXCEPT EVENT corresponds to a new power to terminate an *obligation*, and since this is triggered by a parameter refinement, it *is* covered by SymboleoNLP. The COND_N EVENT pattern class maps to the creation of a new contract termination power, so this is not covered by the mapping aspect of the tool. Addressing these gaps would require a new type of event specification in order to handle the declaration/domain extraction, and a more complex UI, which are left as future work.

The refinement-based pattern classes are verified using unit and integration tests. Once again, we can achieve 100% coverage on our unit tests, and we supplement this with fully-qualified tests that cover all of our pattern class handlers. Listing 5.10 shows a test

Symboleo Specification

```
Domain rentalDomain
  Currency isAn Enumeration(CAD, USD, EUR);
  Landlord isA Role;
  Renter isA Role;
  RentalProperty isAn Asset with address: String;
  SecurityDeposit isAn Asset with amount: Number, currency: Currency;
  PayDeposit isAn Event with deposit: SecurityDeposit, from: Role, to: Role;
  PayAmount isAn Event with amount: Number, currency: Currency, from: Role, to: Role;
  KeepPet isAn Event with agent: Role;
  OccupyProperty isAn Event with occupying_agent: Role, occupied_object: RentalProperty;
endDomain

Contract rental (
  property_address: String,
  currency: Currency,
  extra_amount: Number,
  deposit_amount: Number,
)
Declarations
  renter: Renter;
  landlord: Landlord;
  property: RentalProperty with address := "[PROPERTY_ADDRESS]";
  security_deposit: SecurityDeposit with amount := [DEPOSIT_AMOUNT], currency := Currency([CURRENCY]);
  evt_pay_extra: PayAmount with amount := [EXTRA_AMOUNT], currency := Currency([CURRENCY]), from := renter, to := landlord;
  evt_pay_deposit: PayDeposit with deposit := security_deposit, from := renter, to := landlord;
  evt_keep_pet: KeepPet with agent := renter;
  evt_occupy_property: OccupyProperty with occupying_agent := renter, occupied_object := property;

Obligations
  ob_pay_deposit: O(renter, landlord, true, WhappensBefore(evt_pay_deposit, Date.add(evt_occupy_property, 2, weeks)));
  ob_pay_extra: O(renter, landlord, true, Happens(evt_pay_extra));
  ob_no_pets: O(renter, landlord, true, not Happens(evt_keep_pet));

Surviving Obligations

Powers

endContract
```

Figure 5.13: Updated SYMBOLEO specification

for the WithinTimespanHandler code. A similar test is written for each of our pattern class handlers. All these tests are available in the GitHub repository of this thesis.

Listing 5.10: Pattern class handler test sample

```python
class WithinTimespanEventHandlerTests(unittest.TestCase):
    def setUp(self) -> None:
        self.sut = WithinTimespanHandler()

    def test_handler(self):
        # Initialize a sample obligation
        norm_config = SampleNorms.get_sample_obligation_config('test_id')

        # Fill a pattern class entity with test values
        pattern_class = WithinTimespanEvent()
        pattern_class['TIME_UNIT'] = TimeUnit.Days
        pattern_class['TIME_VALUE'] = 10
        pattern_class['EVENT'] = VariableEvent('evt_test')

        # Run the test
        result = self.sut.handle(pattern_class, norm_config)
        new_norm: Obligation = result[0]

        # Verify the result matches the expected norm
        expected_norm = Obligation(
            'test_id',
```

```
22              None,
23              'debtor',
24              'creditor',
25              PropMaker.make_default(),
26              PropMaker.make(
27                  PredicateFunctionWhappensBefore(
28                      VariableEvent('evt_action'),
29                      Point(PointFunction(VariableEvent('evt_test'), 10, TimeUnit.Days))
30                  )
31              )
32          )
33          self.assertEqual(new_norm, expected_norm)
```

For mapping events to declarations, we have no such evidence-based mapping, and thus we are limited to these rough heuristics. Rather than simply abstracting away this complexity, we allotted some discussion to the topic in order to offer a glimpse into its complexity, as it presents a logical step for future work. Furthermore, we also create a small test suite of full contracts to demonstrate the performance of the tool on entire contracts from a variety of domains. The norm refinements on all of these tests are all extracted as predicted, but we *do* have expected errors arising from the event-to-declaration mapping. This can also provide a starting point for this potential future work.

5.4 Walkthrough

Throughout this chapter, we have discussed the design rationales and the algorithms behind the two main requirements for the SymboleoNLP tool. The first involves enforcing valid CNL entry from the user, and the second involves mapping this CNL into a predictable set of Symboleo refinement operations. We have highlighted this process using the first norm from our sample rental contract. To fully illustrate the functionality of SymboleoNLP, we will now walk through this process with the other two norms:

1. We want to refine 'The renter may not keep any pets on the property [P2]' with something equivalent to '`unless authorization is provided by the landlord`'
2. We want to refine '[P3] the renter must pay an extra fee of $50' with something equivalent to '`In the event of a late payment`'.

We emphasize that the refinements will be *equivalent to* the initial NL in the contract, since we are confined to our CNL, which includes a more restricted way of specifying events. We bring together all of the algorithms that we have discussed (including some from the appendices) in Listing 5.11. The input into this is the initial Symboleo contract specification as well as the specific norm that is being refined (selected by the user). The output is the updated contract specification.

Listing 5.11: Full contract refinement algorithm

```
1 def refine_contract(contract: SymboleoContract, initial_norm: Norm) -> SymboleoContract:
2     # Build the tree using the CNL grammar
3     tree_root = construct_tree()
4 
5     # User traverses the tree, generating the CNL input
6     user_input = generate_cnl_input(contract, tree_root)
```

```
7
8     # Map the pattern class to the required operations, and generate the updated
      Symboleo contract
9     updated_contract = map_cnl_to_operation(contract, initial_norm, user_input)
10
11    return updated_contract
```

5.4.1 P2: Unless EVENT

Consider the norm associated with P2 in its broadened form: 'The renter may not keep any pets on the property'. This corresponds to an obligation to *not* perform the action of bringing a pet onto the property. Framed this way, it is straightforward to capture this as an obligation in S(T), as shown in Listing 5.2. Included in this initial SYMBOLEO specification are the required events, assets, and roles – captured as domain model classes and corresponding declarations. We have the renter role declaration of type Renter, the property asset declarations for 'pets' and 'property', and the evt_keep_pets event declaration of type KeepPets. All of these concepts are used in the ob_no_pets obligation to represent this unrefined norm.

The user will now refine this norm using the CNL by first selecting the parameter that they wish to refine (P2). This initiates the CNL generation using the tree-traversal method. The UNLESS unit is a child of the root, so the user is able to select it. It is a static unit that simply corresponds to the value 'unless'. The only child of this unit is the EVENT unit. Since they will be entering a new custom event, the user will then select the CUSTOM_EVENT unit. The original refinement contains the event 'authorization is provided by the landlord'. This refinement is in *passive* form, so it cannot be identically replicated using our CNL. We must convert it to the *active* form: `'landlord provides authorization'`. This is accomplished by the user entering values for the SUBJECT ('landlord'), the T_VERB ('provides'), and the DOBJ ('authorization'). All three of these are dynamic units. There is also somewhat of an implied co-reference on the original refinement. The authorization is clearly 'for pets', but we would like to make this explicit: 'landlord provides authorization for pets'. The user would therefore continue by filling out a PREP_PHRASE unit with the value 'for pets'. This event is now properly specified within our CNL and is synonymous with the original statement. The result is a list of user inputs that correspond to the pattern class P_EXCEPT EVENT, which is passed to the mapping step. These inputs are found in Table 5.4.

	Input Unit	**Type**	**Value**
1	UNLESS	static	unless
2	EVENT	empty	
3	CUSTOM_EVENT	empty	
4	SUBJECT	dynamic	landlord
5	T_VERB	dynamic	provides
6	DOBJ	dynamic	authorization
7	PREP_PHRASE	dynamic	for pets

Table 5.4: Input units for P2 refinement

The goal of our mapping step is to produce the ContractUpdateObject, consisting of the required SYMBOLEO update operations (or *equivalent* operations):

- **Norm:** `pow_allow_pets: Happens(evt_allow_pets) -> P(renter, landlord, true, Terminated(ob_no_pets))`
- **Declaration:** `evt_allow_pets: AllowPets with provider := landlord, allowance := pets;`
- **Domain Event:** `AllowPets isAn Event with provider: Role, allowance: Asset;`

First we perform the pattern extraction step, which will yield our fully-specified pattern class object. This includes extracting a CustomEvent entity which represents the act of providing the authorization. Next, we extract the norm update. The P_EXCEPT EVENT pattern corresponds to adding a power to terminate an existing obligation. Creating the power is predictable, as we only need to reference the initial obligation ('ob_no_pets'). We re-iterate that this predictability was precisely the justification for the inclusion of the SYMBOLEO operation for adding a new norm. We must also add a *trigger* to this power, which references the new event. The event reference is built using linguistic entities from the CustomEvent object. The new and fully-specified power is added to our ContractUpdateObject.

Next, we extract the domain updates. We first extract the event declaration by joining the verb lemma and direct object head into the event name 'ProvideAuthorization'. Then we analyze each of the noun phrases in the event and heuristically extract the event slots. The subject ('landlord') corresponds to a role. We can therefore map this to a 'providing_agent' slot. The direct object ('authorization') maps to the 'provided_object' slot. The prepositional object ('pets') maps to the somewhat vague 'provided_details' slot. This highlights the type of error associated with our simple heuristics discussed previously. The declaration is given the identifier from the previous step ('evt_provide_authorization)', and the domain event is then created from this declaration. We can compare the expected operations above with the actual operations below to get a sense of the types of errors involved in mapping an event to a declaration. In some cases, the difference is acceptable since the terminology is synonymous.

- **Norm:** `pow_terminate_ob_no_pets: Happens(evt_provide_authorization) → P(renter, landlord, true, Terminated(ob_no_pets))`
- **Declaration:** `evt_provide_authorization: ProvideAuthorization with providing_agent := landlord, provided_object := authorization, provided_detail := pets;`
- **Domain Event:** `ProvideAuthorization isAn Event with providing_agent: Role, provided_object: Asset, provided_detail: Asset;`

5.4.2 P3: Added Antecedent

Our final unrefined norm, 'The renter must pay an extra fee of $50', corresponds to a simple payment obligation along with its required roles, assets, and events. The target refinement is a conditional statement equivalent to `'In the event of a late payment of the security deposit'`. The user will choose the third parameter to refine and be presented with the child nodes of the root. One of these is the static unit 'in the event that', which is selected by the user. Next, they will need to specify an event. The original event is 'late payment of the security deposit', and we want to use our CNL grammar to represent something similar. With the current CNL, potential options may include 'in the event that renter pays security deposit late' or 'in the event that renter violates payment of security deposit'. There may be many other event expressions in pure NL that would be synonymous, but we are restricted to our CNL. We will use our first option, and our input will correspond to Table 5.5.

	Input Unit	**Type**	**Value**
1	IN THE EVENT	static	in the event that
2	EVENT	empty	
3	CUSTOM_EVENT	empty	
4	SUBJECT	dynamic	renter
5	T_VERB	dynamic	pays
6	DOBJ	dynamic	security deposit
7	ADVERB	dynamic	late

Table 5.5: Input units for P3 refinement

Following the input generation, the following familiar steps are taken:

- **Pattern identification:** The input list is identified as the COND_A EVENT pattern.
- **Event Extraction:** Since this pattern class is event-based, we must extract a CustomEvent from the input list.
- **Norm Update Extraction:** The pattern class corresponds to the addition of an antecedent to the existing norm, which is created by a specific CondAEventHandler class, and added to the ContractUpdateObject.
- **Domain Update Extraction:** Since we have a new event, we must identify an event declaration and its properties, and then deduce a new Domain Event. These artifacts are added to the ContractUpdateObject. Since both noun phrases (renter and security deposit) are referenced from the existing set of declarations, they do not need to be added as new assets.

The result of these steps is another populated ContractUpdateObject, which is then applied to the existing contract. This consists of the following:

- **Norm:** `ob_pay_extra: O(renter, landlord, Happens(evt_pay_deposit_late), Happens(evt_pay_extra))`

- **Declaration:** `evt_pay_deposit_late: PayDepositLate with paying_agent := renter, paid_object := deposit;`
- **Domain Event:** `PayDepositLate isAn Event with paying_agent: Role, paid_object: SecurityDeposit;`

A disadvantage of this approach is that it requires the user to restate the existing event of paying the security deposit, and it results in a similar-looking event (evt_pay_deposit vs evt_pay_deposit_late). Another interpretation of this refinement would be to consider this to be a *violation of an existing obligation* rather than a brand new event. This is the interpretation suggested by the initial Symboleo specification in Listing 5.1, where we have the predicate `Happens(Violated(ob_pay_deposit))`. It may be beneficial to include the ability for the user to reference existing events and norms as they construct their refinements. Another option may be to build an event-matching module, which could detect similarities between a new and existing event, preventing the creation of unnecessarily similar events. There may be many other UI-related options for achieving this goal, and these ideas may be implemented in future work.

With these two examples, and the first norm refinement discussed in the previous sections, we have now recreated a Symboleo specification that is synonymous with the original, found in Listing 5.1, using the SymboleoNLP tool. A contract author could therefore use the tool along with this template to create a variety of different CNL refinements and corresponding Symboleo specifications. Once the contract author is done refining the NL parameters, they can download the corresponding Symboleo specification for use in other smart contract applications.

Chapter 6

Discussion

Throughout this thesis, we have accumulated a set of results and contributions, while discussing various threats and limitations, many of which have been translated to concrete ideas for future work. In this chapter, we will bring all of these ideas together into a more focused discussion. We will summarize and discuss our key contributions, as well as the threats and limitations associated with these results, first for the CNL and RQ1-RQ3 (Section 6.1) and then for the SymboleoNLP tool and RQ4-RQ5 (Section 6.2). Finally, we will discuss some more long-term goals of this research, and ideate on a path towards those goals in light of our research as well as other related technologies, including Large Language Models (Section 6.3).

6.1 CNL for Contract Refinements

6.1.1 RQ1: Identifying Operations and Symboleo

The goal of our first three research questions was to construct an evidence-based CNL for contract refinements in Symboleo, a specification language for legal contract monitoring and execution. Since our the end goal is to perform a series of update operations against an existing Symboleo specification, our first RQ involves identifying the range of these operations. We are framing this work as an extension of a template-based approach, which typically involves filling in blanks with simple values, such as dates and names. We seek to extend this traditional FITB to accommodate a larger variety of customizations, but since we are still using templates, we maintain the idea that template customization involves *refinement*. We therefore want to identify any operations that refine a Symboleo contract. We began by primarily considering the *syntax* of Symboleo, and defined a refinement operation as one which *adds* information. Examples of this include various ways of refining the `Happens` predicate, adding an antecedent or trigger, and adding and modifying various other components of the domain model.

Symboleo is a precisely-defined language and therefore the number of potential operations are finite. We can therefore aim for the list of refinement operations identified in this step to be *exhaustive*. Throughout the rest of our research, we did not identify

any other operations that were missed, but we note that we identified certain *variations* on existing operations. For example, we initially eliminated operations that add brand new norms on the basis that this goes against a template-based approach. However, certain norms involve very predictable components, such as powers that reference existing obligations or the contract itself. These were only detectable when performing *semantic* analysis using real contracts (RQ2 and RQ3). We leave open the possibility that other specific variations of refinements may emerge upon analysis of larger datasets. Furthermore, we limited our operations to those that directly affect the norms of the contract (powers, obligations, surviving obligations) and the domain model. This does not include additional entities specified in Symboleo, such as the pre-conditions, post-conditions, and constraints. Future work may involve incorporating refinements on these components of a Symboleo contract.

We also note that Symboleo is under active development, and additional constructs may be added as the language grows. In this work, we have used a specific version of the XText grammar of Symboleo, but we have noted a few areas where there may be room for improvement, and we will summarize the main ones here:

- **Frequency:** It is fairly common to see obligations related to *frequency* in NL contracts, for example, for recurring payment schedules. Incorporating frequency and recurrence may require the introduction of a special type of trigger into the language.
- **Weak predicates:** The `WhappensBefore` and `WhappensBeforeEvent` predicates are *weak* in the sense that the referenced timepoint is *not guaranteed* to happen, which means the contract cannot be guaranteed to terminate. This may motivate a decision to remove weak predicates from future versions of Symboleo. From a NL contract generation perspective, this may imply that any temporal refinements include some fixed anchor date in case the referenced event does not come to pass. We also noted the possibility of introducing a refinement from a weak predicate to a strong predicate.
- **Negated Events:** For an event-based contract monitoring system, we are assuming that some service will alert the contract that the event has come to pass. For example, if we want to know if the temperature in a meat delivery truck exceeds a certain temperature, we can assume that a connected temperature sensor will be able to send a notification, and the contract will perform the prescribed behaviour. However, we have seen several `not Happens` predicates in this work, and these are conceptually more difficult to monitor. If we are monitoring for the *non-occurrence* of the event, then this might imply that we would need to constantly be polling some sort of event log to ensure the event has not occurred. The frequency of the polling period would correspond to an imprecision with which the events are monitored. For these reasons, future iterations of Symboleo may remove certain types of negated predicates.
- **More Legal Concepts:** We briefly discussed the possibility of including *rights* in our analysis, and there are relevant relationships between various types of legal positions [47] which may motivate future incorporation into Symboleo. This is

especially relevant when we are considering *refinements*, as certain types of linguistic refinements (e.g., unless EVENT), might change one legal position into another legal position.

- **Symboleo Situation:** The current specification of the `Occurs` predicate requires a `Situation` and an `Interval`. A `Situation` currently includes only states for existing obligations and powers (or the contract itself). While we can work around this by framing NL situations as *negated events*, future work may involve expanding the expressiveness of the `Situation` construct.

Some of these changes to the Symboleo specification language would reduce the expressiveness of the language, thus rendering it incapable of representing certain contract scenarios. As work progresses towards the implementation of real smart contracts that monitor for events and perform certain operations, these constraints on expressiveness may become clearer. This is not necessarily a *defect*, as it may result in Symboleo becoming a specialized fit for a particular domain of legal contracts. Preliminary testing has been done on using our generated Symboleo artifacts with other tools in the Symboleo ecosystem, such as the code generator [73]. While initial results are promising, further testing on a wider variety of contracts needs to be done. We may find that certain types of Symboleo constructions, such as those mentioned above, may not translate to realistic contract monitoring situations. In this work, we have explored *many* types of contracts, as we want to limit our assumptions about the potential use cases of Symboleo for the time being, but we have paid special attention to cases that are more amenable to automation (focusing on payments, scheduling, etc.). As the Symboleo language evolves, the results of this thesis will need to be updated, and the methodology for doing so begins with RQ1, where we decide *which* Symboleo operations we want to include in our contract refinement system.

6.1.2 RQ2: Semantic Analysis

Defining the potential refinement operations in RQ1 provides us with the full range of what will ultimately be possible with a tool such as SymboleoNLP. In a sense we can think of it as the *output* possibilities. If we could assume that contract authors were fully versed in Symboleo, we would need to go no further than RQ1. A contract author could be presented with a Symboleo contract, and make the desired customizations in Symboleo through a well-constructed UI. Following the PENS classification schema [55], this would mean high precision and simplicity, but unacceptably low expressiveness and naturalness. The point of the CNL is that it should feel natural for a user, and this is why we take an evidence-based approach to the construction of the language; we looked at real NL contracts to empirically determine what linguistic patterns are considered *natural* in the legal domain. While the empirical analysis is the domain of RQ3, determining which patterns to look for *in the first place* is the goal of RQ2.

Before deciding on these patterns, we first introduced the principle of integrity, which states that we want our NL template T to *always* be grammatically correct *and* we want the corresponding Symboleo specification S(T) to *always* be valid. This principle allows

for broadly-defined contracts, and limits us to refinements that take the form of linguistic adjuncts. NL contracts may not be written with formalization in mind, so it may be possible for refinements to be *implicit* in the text of the contract. For example, Symboleo is not concerned with the explicit representation of a *permission*; however, a permission for one party may correlate with an obligation for another party. Performing an assessment for all *types* of refinements (not just explicit adjuncts) would be a valuable future step towards validating the usefulness of this principle.

We are thus aiming to identify optional refinement patterns in NL that correspond to the operations identified in RQ1. We approached this by first considering the formal semantics of Symboleo [68]. These definitions suggest some trivial NL refinement possibilities (e.g., refining a `Happens` to `ShappensBefore` could correspond to before DATE). In order to increase expressiveness and naturalness, we grew these potential patterns by identifying synonyms for targeted keywords with thesauri and using generative language models (e.g., ChatGPT). The result was a mapping between linguistic pattern classes and the operations defined in RQ1. If one of these pattern classes is entered as a refinement on a broad contract norm, then we can predict what the resulting Symboleo operation will be.

In addition to the limitation on refinements imposed by our principle of integrity, there are a few other points worth mentioning. Since Symboleo deals with monitoring *events* of contracts, we focus on refinements that impact the occurrence of these event, namely temporal and conditional refinements. This excludes other types such as locative or instrumental, though we have provided some potential linguistic patterns for these types of refinements as a starting point for future work. In taking a semantic analysis, we also found further limitations imposed by Symboleo itself, such as the exclusion of provisions for frequency. We noted there are certain adjuncts related to frequency that might fill this role, and this could be addressed in future work if the concept of frequency is introduced to the Symboleo language.

6.1.3 RQ3: Evidence-based Construction of the CNL

The result of RQ2 is a working CNL for contract refinements that apply to Symboleo, consisting of a dynamic EBNF grammar and a set of mappings between pattern classes and Symboleo operations. The construction of this CNL was largely done heuristically by looking at *general semantics*. This provides us with a starting point for a CNL and allows us to explore important linguistic concepts in greater depth. However, we must always keep in the mind the endless complexity associated with NL, and acknowledge that there may be many types of refinements that were *missed* from our generic semantic treatment. We therefore want to build our CNL using evidence from real contracts, and this is the goal of RQ3. We outline a detailed process for constructing a CNL from a dataset of real contracts. For each of these steps, we have discussed sources of threats to validity. We can categorize these threats as internal threats or external threats. Internal threats are those which may compromise any causality that is extracted from the research. We are trying to identify linguistic patterns of refinements from real contracts to find what Symboleo operations those refinements will correspond to, so an internal threat would be anything that might cause us to question the link between a pattern class and a

Symboleo operation. External threats are those which might affect the ability of the results to generalize.

A key source of these threats is our dataset. Since this is relatively novel work focused on a specific language (Symboleo), there are no existing datasets for this work, so we must create our own. Our initial dataset comes from a single source (CUAD), is of a limited size, and was filtered to focus on smaller contracts to facilitate the tedious manual labelling. All of these factors pose potential threats. A larger dataset that is annotated by multiple experts in Symboleo and the legal domain would provide higher reliability that we have identified the proper mappings between NL refinements and Symboleo. Pulling contracts from other *sources* in addition to CUAD may improve generalizability, depending on the specific domain that we are interested in. The contracts required by the SEC and those subsequently selected for CUAD may have some implicit structures that are not found in other contracts. It may be unsafe to assume that this work will generalize beyond this dataset.

Another important source of threats comes from the heuristics we used to find refinements. Our initial dataset resulted in about 2700 sentences potentially containing norm refinements. In order to extract these 2700 sentences, we employed keyword heuristics to identify potential obligations. These heuristics were based on keywords found in relevant related work, but there is still the possibility of false negatives, where we may have omitted a sentence from this set of 2700 because an obligation was framed differently. This also highlights a major assumption made by this work – that a given norm is *wholly contained* in a single sentence. This is often not the case in reality, where certain conditions and exceptions to a norm may be stated in subsequent sentences, in a longer structured list of exceptions, or possibly even by referencing a completely different document. It is possible that an extracted norm from our dataset is *incomplete*, since there may be sentences throughout the rest of the document that qualify the initial norm statement.

Our next set of heuristics comes from the fact that we want to identify refinements that will correspond to Symboleo. For this we also use a set of keyword and pattern *heuristics*. These heuristics have a more solid foundation, as this was an explicit goal of RQ2. The linguistic patterns found in RQ2 provide us with keywords and patterns that we can look for in larger contract datasets. There remains a possibility for false negatives, and one mitigation strategy we use is to look for broader linguistic structures *beyond* our heuristics from RQ2, such as subordinating conjunctions and prepositional phrases. Many of the patterns from RQ2 were in the form of specific prepositional phrases, but this strategy would mean looking for more general prepositional phrases to see if they contain relevant refinements. These pattern searches will in turn rely on the tools used to *find* such patterns, such as SpaCy. These pattern finders are probabilistic, so it is possible that they fail to properly identify a prepositional phrase that may correspond to a refinement.

6.1.4 Classifying our CNL

Kuhn [55] points out that there are often trade-offs between the PENS factors in CNL design, and this is indeed what we find throughout our CNL construction process. We began with the precise and relatively simple formal specification language of Symboleo,

and made it more expressive and natural using linguistic analysis and empirical analysis of real contracts. By making the language feel natural and expressive, there is potential to reduce the precision and simplicity. The goal is to keep these reductions acceptable for our purposes.

Lowering the precision means that we introduce ambiguities in the mappings. For example, a potential case of ambiguity in our CNL is the after EVENT pattern, which can result in a conditional refinement or a temporal refinement. The ambiguity is easily resolved, however, by considering the context of the refinement (if it's in an existing conditional statement, then it's a temporal refinement. If no condition exists, then it's treated as a new conditional refinement). We have seen a few other cases of ambiguities in our CNL, but in each case, we have offered and implemented strategies to resolve them. If we were to encounter an ambiguity that was too complex to resolve, we may choose to exclude it from the CNL or come up with more sophisticated heuristics. Lowering the simplicity corresponds to more patterns and functionality, which in many cases can be framed as an implementation challenge. If the language becomes too complex, tools such as SymboleoNLP that support the CNL may become harder to maintain.

As for expressiveness, our goal is to ensure that we have at least one NL pattern for each type of refinement operation identified in RQ1. This gets somewhat more complicated when we see how there are different ways of expressing certain concepts in NL, and each of these expressions have different *meanings*. For example, 'between Monday and Friday' and 'for 2 weeks following the agreement date' are different ways of expressing an interval refinement but the linguistic patterns must be used in different NL contexts. These different contexts were identified through our CNL construction, and introduced new needs for expressiveness. We leave open the possibility that other ways of expressing concepts such as an interval will emerge upon further contract analysis, motivating the expansion of the CNL.

A variety of factors must guide the decision to include other expressions of a Symboleo concept in the CNL. For example, we have noted that the expression of an interval may be spread out over a sentence and not easily isolated as a refinement. While this could be added in future work, it may present major *complexity* challenges. In other cases, increasing the expressiveness is relatively trivial, and involves simply adding a synonymous keyword to a pattern class. For example, the before DATE pattern already covers the `ShappensBefore` refinement, so we technically do not require prior to DATE in the CNL, since it has the same meaning and mapping. However, this latter pattern contributes to increased naturalness and expressiveness with acceptable costs to precision and simplicity, so we include it.

6.1.5 Contributions

The most tangible contribution from this discussion is the evidence-based CNL itself. This CNL directly provides us with a set of requirements for a proof-of-concept tool, such as SymboleoNLP. An equally important contribution is the methodology of constructing the CNL itself. While we have identified a number of important threats to this work, having a clear methodology is important towards future iterations of this construction

process. With a detailed process laid out, and the key threats identified, subsequent iterations can be more easily carried out to obtain a more reliable and scalable CNL by addressing some of the threats that we have discussed. While some steps in the methodology are necessarily manual, other parts have been coded in Python, and another area of future work may involve further automation of some of these steps. For example, we may be able to construct a mapping between specific patterns and the more general *pattern classes* to make this process easier. However, as this is novel work, many of these processes should include *some* degree of manual analysis so that other potential threats can be identified.

Our methodology includes a simple evaluation metric that is measured against a separate test set, whose creation is also included in our process. While our pattern coverage of 0.97 (Sec. 4.5) is promising, this is still a fairly small dataset, and for now we put more emphasis on the qualitative aspect that analyzes the cases from the test set that *missed*. If this process becomes more automated, it may be beneficial to perform n-fold validation, where we construct n different construction/test sets from our data, and average n different accuracy measurements.

In addition to the CNL and methodology, a related contribution is these manually labeled artifacts, which have potential to act as datasets to other work related to Symboleo. We were specifically looking for requirements that aligned with our identified Symboleo operations from RQ1. However, it is worth pointing out that Symboleo is built on an ontology, and may therefore share commonalities with other specification languages built on similar concepts. It therefore may be possible that the methodology and the resulting artifacts may be applicable outside the Symboleo ecosystem.

In summary, we have identified three important contributions arising from our RQs: An evidence-based CNL, a methodology for constructing the CNL, and a labeled dataset that may be applicable to further related work. The main source of threats identified include a limited dataset, a lack of expert knowledge, and false negatives arising from heuristics. We have discussed ideas for addressing these threats as part of future work. Furthermore, we have some self-imposed limitations, since we've constrained the problem to specific types of refinements. Future work may involve relaxing some of these limitations to address more diverse refinements.

We also want to posit what an *ideal* CNL construction scenario looks like. We would begin with a large set of contracts that are applicable to smart contract scenarios. These contracts would come from a variety of sources to eliminate any bias associated with any single source. These contracts would be manually inspected by a team of experts to extract refinements that correspond to Symboleo norms. This labelling process would be designed so that any given document is seen by multiple labellers, so that we can have a high degree of inter-annotator agreement. Once we are satisfied with our dataset of refinements, we can split the data into a construction set and a test set. We would then run through the rest of our procedure, identifying the basic patterns and the more general pattern classes of the refinements in the construction set. Ideally some of these steps would be further automated. From this we would produce a CNL, and we could test the coverage using the test set. This construction/testing process could be performed multiple times using n-fold cross validation to get a reliable accuracy metric. The entirety of the dataset could be used for the final CNL, for application to a tool such as SymboleoNLP.

6.2 Proof-of-Concept Tool using the CNL

SymboleoNLP is a tool that applies a CNL for contract refinement to real contract templates. At this stage, we have constructed a proof-of-concept tool to demonstrate its functionality, and highlight potential challenges that would need to be addressed if such a tool were to scale up to production code. We identified two key requirements that follow directly from the construction of our CNL: Ensure that the CNL can be entered into the tool, and ensure that the mappings between the inputs (CNL) and outputs (Symboleo operations) are correct. We framed these as our research questions RQ4 and RQ5, respectively.

6.2.1 RQ4: Generating Valid CNL

For the first requirement, we used a tree-traversal approach to allow the user to input a predictable sequence of *unit types*, which can guarantee that we cover *all* of our pattern classes and *only* our pattern classes. However, since the grammar is *dynamic*, the user has the option of entering invalid input (bad syntax and/or bad semantics) into the system. This is the first source of potential error. This will be a feature of any application that seeks to allow expressive user input, and we acknowledge that any solution will be probabilistic and/or heuristic in nature. We have outlined two potential mitigation strategies. First, we can acknowledge the fact that we are working within a specific legal contract domain, and therefore it is likely that the content of the input text will be related to this domain. We therefore provide the user with *suggestions* about what to enter. For example, when the user is prompted for a noun phrase, we suggest the set of roles and assets from the domain model. The second strategy is to integrate modules that could perform relevant validation against user input. For example, if the user chooses to enter a transitive verb, we might have a module that first checks that the input is indeed a real verb (syntax), then another may check if that verb is *transitive* and that it makes sense in the context of the contract (semantics). The former module would likely be much easier and less prone to error than the latter. We have designed the code using object-oriented principles, facilitating the easy integration of such modules in the future.

Another drawback to our approach is that we make an assumption of linguistic knowledge on the part of the contract author. For example, they need to understand concepts such as subject, predicate, and verb types. Even if this were a valid assumption, it would nonetheless cause a degree of friction on any type of UI that implements the functionality. Furthermore, the definitions of many linguistic concepts such as predicates and prepositional phrases are not universally agreed-upon. Ideally, we might provide a more user-friendly freeform input as a user interface, with the details of the tree-like structure of the grammar hidden from the user. This would involve parsing the input units out of potential text. A simple approach might be to split out the words based on the spaces in the text. We iterate through the words, check which unit type each one might correspond to, and try to find if any pattern classes match the structure. The difficulty with this is that phrases that function as a single unit (e.g., a noun phrase) can be made up of many words. Once again we run into the complexity problem of NL, and so any solution will be probabilistic.

For RQ4, we can therefore state that generating a tree from the EBNF grammar component of our CNL can guarantee that the *types* of input will adhere precisely to our CNL and only our CNL. This requirement is evaluated through the use of a multi-layered test suite consisting of unit tests and larger tests on fully-specified contract scenarios. Due to the dynamic nature of our CNL, there are multiple potential sources of user error, such as the ability for the user to enter ungrammatical input. These may be mitigated through UI design, introducing contextual knowledge, using heuristics, or using probabilistic NLP modules for validation.

6.2.2 RQ5: Mapping CNL to Operations

Norm Updates

The second requirement of our tool is to map a given CNL refinement into the proper Symboleo operations as defined by our CNL. There are two parts to this mapping – extracting norm updates (e.g., mapping `Happens` to `WhappensBefore` with the proper arguments) and extracting domain updates (e.g., adding newly created events and their declarations). Our first three RQs addressed the character of the norm refinements, so the implementation of the norm update extraction was fairly straightforward. We use unit tests and integration tests with full coverage to increase our confidence that all norm updates are processed correctly. In cases where a pattern class can map to more than *one* potential operation, we need to look at the context of the norm being updated to resolve the correct operation.

Domain Updates and Events

The other mapping sub-process, the domain update extraction, presents many more threats, which largely stem from the complexity involved in specifying events. While the event specification in our grammar can cover a wide range of events, it is still quite minimal considering the countless ways that events can be specified in the entirety of NL. This limits our ability to represent completely real contracts in our system. In this thesis, we have restricted our event examples to those that *do* meet this limited specification, or we have simplified real events to fit with our specification mold. The restrictions are modeled off of existing event specifications outlined in Section 3.2.

Given such an event, as specified by our CNL grammar, we need to identify the *slots* that give *meaning* to the event, and then fill those slots with the proper values. This involves a set of rough heuristics based on details of the event components. While there is room for a variety of mappings between events and declarations (e.g., occupier is synonymous with occupying_agent), the heuristic approach can lead to less semantically precise results for our declarations and new domain classes, so we note this as a higher priority for future work. This future work may incorporate further frame semantics, or it may involve a more rigorous process, similar to our CNL construction, which we will now discuss.

If we precisely labelled all of the events contained in our set of refinements as well as their grammatical components, we could generate a more relevant EBNF grammar

for contract events. This labelling process would also involve manually identifying the domain model updates associated with each of these events. For example, suppose we have the event 'the seller delivers the goods to the buyer at their warehouse'. We could perform a linguistic parse on this sentence using NLP libraries such as SpaCy to get the various linguistic components (parts of speech, dependencies, entities, etc.). We would manually link these to event declaration components (e.g., the subject 'seller' maps to delivery_agent). We would then look for more general patterns in this mapping (e.g., mapping the subject to {lemma}_agent). Similar to how we built a dataset that links temporal/conditional refinement patterns to norm updates, we would have a dataset that links event specifications to domain classes and declarations. This could then be used to construct a related CNL mapping that could be implemented in SymboleoNLP. This scenario would require much less reliance on the heuristics introduced for our domain update extraction. If such a process were undertaken, a metric of event specification coverage could similarly be calculated and analyzed. If the labelled dataset were to grow large enough, we might be able to use more sophisticated machine learning techniques to build a prediction model. Some specific complexities that could be incorporated through this more rigorous process might include subordinate clauses, conjunctions/disjunctions, and more sophisticated conditional statements (e.g., '...before event A or event B, *whichever comes first*').

For RQ5, we can therefore state that we have addressed the *norm update* component of the Symboleo operations, which handles the temporal and conditional refinements on a norm. This is validated through the use of a rigorous test suite. The corresponding *domain update* component of our Symboleo operations is largely based on rough heuristics, and a high priority for future work would be to apply a similar treatment to the specification of events that we applied to the refinement patterns.

6.2.3 SymboleoNLP Road Map

We will now summarize a list of recommendations needed to make the SymboleoNLP tool fully usable as a production tool. We will call this idealized version of the tool SymboleoNLP_1 for future reference. The prerequisite to this tool is an accurate and reliable CNL, which should be constructed using the methodology described in RQs 1-3. We can add onto this our proposed similar method for constructing a more precise event specification. Before the tool can be used, there is a detailed manual pre-processing step that must be undertaken by Symboleo experts, which was discussed in Section 5.1. This pre-processing involves identifying valid refined norms, and broadening them. There is potential for some of this pre-processing to be automated, such as the identification of the broad norms, and perhaps modules that can *assist* in extracting the rest of the required templates.

For our first requirement of valid CNL generation, improvements would address the potential for syntactic and semantic errors by injecting validation modules into the flow. A freeform input in front of the tree-traversal may eliminate the assumption of linguistic knowledge. We emphasize that these solutions would be largely probabilistic and may require rigorous evaluation on a module-by-module basis. A key contribution of this tool is the implementation of the tool's *structure*, which reduces other types of complex

NLP functionality to specific modules, which can be easily injected into the code. For our second requirement, the norm update extractor would expand in response to any changes in the constructed CNL. This would involve adding more pattern classes, handler functionality for each pattern class, and any corresponding tests. The domain update extractor would be more closely linked to the proposed evidence-based event specification, replacing the rough heuristics that we have introduced.

We also note that this entire tool would require a usable UI. Our proof of concept includes a web UI demonstration of the tool, but it may also come in the form of a desktop app or other type of software. Regardless of the medium, an important factor would be the NLP functionality. Some of the proposed modules may rely on sophisticated NLP techniques, which requires large computing resources and can take a long time. This fact may motivate the use of asynchronous message queues within the architecture, where the main processing takes place *off* of the main processing thread. Usability of the tool could be evaluated in an experimental setting, where we compare the process of using the tool with a fully manual process. This type of testing is seen in related work, and requires careful experimental design [53, 84]. The usability of a tool may also be highly dependent on front-end design considerations, which is an entire industry unto itself, involving knowledge of human psychology and rapidly-changing front-end technologies.

6.3 Longer-term Road Map

Now that we have proposed ideas for concrete future work for both the CNL and the SymboleoNLP tool, we want to look even further ahead towards the longer-term goal of full formalization of NL contracts, which is one of the key motivating factors for this work.

At a high level, a full formalization process may consist of the following steps:

1. The user inputs a fully-specified NL contract, perhaps in PDF form, into the application. There could also be a separate module that allows the user to create a contract from a pre-defined template. Either way, our main input is a full NL contract.

2. The user specifies an output format. This may be a specification language such as Symboleo, or perhaps it is an executable JavaScript implementation of a Smart Contract.

3. Based on these inputs, the application may ask the user a series of clarifying questions. Perhaps it will ask about the intended end goal, ask the user to clarify unresolved NL ambiguities (such as co-references), or clarify the structure of the desired output.

4. The tool will output a folder containing a set of generated artifacts. With Symboleo, it may generate a single Symboleo contract specification. This could then be used with other applications within the Symboleo ecosystem.

6.3.1 Towards Full Formalization

Supposing we take our ideal tool described in the previous section as the starting point (SymboleoNLP$_1$), and take this oracle-like application as our endpoint (let's call it SymboleoNLP$_F$), we can theorize about the steps needed to get from one point to another. One way to think about this gradual process is to frame it in terms of expanding the range of inputs and outputs. SymboleoNLP$_1$ is able to handle specific types of refinements as defined by our CNL. The starting point for our CNL was in RQ1, where we defined the desirable operations (i.e., the output). Therefore the first step may be to expand this list of operations by relaxing some of the restrictions we imposed. For example, perhaps we begin by allowing the creation of all powers. This would be a logical starting point, since the consequents of powers contain references to other norms (or the contract itself). Once a particular restriction has been relaxed to allow new operations, we then look for the semantic alignment with these operations (RQ2). For example, with powers, we may look into the various semantic ways of expressing powers or rights. We may also want to consider how powers could be *implicit* within contracts. This would give us a set of heuristics that we can apply to a real contract dataset (RQ3). We can run through our dataset analysis procedure to construct a new and more flexible CNL. At this point, our methodology may remain fully intact and scalable. Each time we expand our CNL, we would want to include the analysis of metrics to ensure that we are still able to reliably map linguistic patterns with Symboleo operations. If this fails at any stage, then it may indicate that a larger dataset is needed.

Suppose we have successfully integrated the operation of adding a new power. Our CNL has expanded accordingly and we still have a respectable accuracy metric on our CNL. Further restrictions that we may want to relax next might include:

- Adding new obligations
- Adding new roles
- Addressing pre-conditions, post-conditions, and constraints

Essentially we are describing the process of gradually relaxing all restrictions from RQ1 until we are capable of *any* Symboleo operation. Each step would be accompanied by a methodological re-construction and evaluation of the CNL, as well as an evaluation of the usefulness of the tool. Another restriction that would eventually need to be relaxed is our principle of integrity. We would no longer restrict refinements to be those that are optional. This would result in further expansion of our CNL.

As we relax these restrictions, our CNL would eventually get very large. As we accommodate more complex event types and larger linguistic structures, any probabilistic modules used in the process may compound into unacceptable error levels, which would be caught in the tool evaluation process. Authors might try to use these later versions of the tool, but find that it is easier to use earlier versions of the tool that have less flexibility but more reliability. We may face further issues of ambiguity where a single pattern class may correspond to a wide variety of operations depending on the context, and the resolution of these operations may be reduced to probabilistic estimates. It is

at this point where we may reach the limits of our proposed methodology, as well as the limits of deterministic NLP.

Since our output range (i.e., Symboleo operations) is finite, we may still maintain a list of mappings between patterns and operations. However, the input will have become too complex to represent as an EBNF grammar. We now want to capture the *entirety of NL* as potential input. We have discussed the ability of gradually integrating various probabilistic modules into the flow, but until this point, these modules have been constrained to fairly precise tasks (e.g., syntax validation, determination of verb type). Now, however, we apply it to the much broader task of identifying a Symboleo operation from unrestricted NL. This is where we may want to begin exploring newer technologies such as sophisticated Large Language Models (LLMs)s based on a transformer architecture (e.g., Generative Pre-Training Transformer (GPT), BERT). The labelled datasets created throughout earlier iterations of development may be repurposed as training data for an LLM to help it learn these mappings.

6.3.2 Incorporating LLMs

The writing of this thesis comes at an exciting time in the field of NLP. Transformer-based architectures are excelling on many NL tasks, and LLMs such as ChatGPT are being made widely accessible. While these are indeed exciting, there are also many reasons to be cautious. These models are trained on huge amounts of NL data, and they are largely built to generate the text that is *most probable*, not necessarily what is right. Some problems associated with these LLMs include hallucinations [60] and the potential for model collapse [81]. A hallucination is when the model generates text that is incorrect or non-sensical in an attempt to answer a question. This issue can be compounded if the model appears *confident* in its own hallucination. Model collapse describes a scenario where a model *trains on its own data* (since it is unable to distinguish its own content from *actual* human-generated content), and it degrades the performance. The field is still quite young, and it may be possible to fully or partially address these and other issues, so it is worth considering how these technologies could be incorporated into the problem laid out in this thesis, either in the near or far future.

LLMs as Modules

We have carved out a process of formalizing refinements made with our CNL, but within this structure, we have identified key sources of error, which may be mitigated by introducing probabilistic modules. These error sources present opportunities to integrate an interface to a powerful language model. Some functionality that we have noted includes identifying syntactic correctness, semantic correctness, and identifying the verb type entered by the user. As the event specification gets more complicated, further functionality needs will likely be identified and integrated. An LLM may be overkill for something like *syntactic* correctness, but detection of other errors is more complicated. For harder problems, we can imagine an API that takes part of the user input, and formats it into a valid *prompt* for the LLM (e.g., 'Is the verb in the following sentence transitive or intransitive: ...?').

This example is a classification problem, so the response can be easily validated and processed accordingly.

Any module that is integrated would require proper evaluation, and an important drawback about these modules is the lack of explainability. This trade-off between explainability and performance is not an unusual occurrence in NLP tasks, and this is an important consideration when applying these models to domains that may require high explainability (health, legal, etc.). Explainability may not be a major concern if the LLM is being used for a well-defined sub-task such as verb type detection. A bigger problem may simply be reliability of the LLM. Fig 6.1 shows an example where ChatGPT fails to answer a simple question about the verb type in a relevant sample sentence. It is likely possible to fine-tune instances of these models on specific tasks, but this highlights the fact that LLMs may not be a panacea for all NLP tasks at this point in time. Chen et al. [19] apply GPT-4 to the related field of goal modelling, and through rigorous experimentation, note that it can output a wide variety of different responses to the same question.

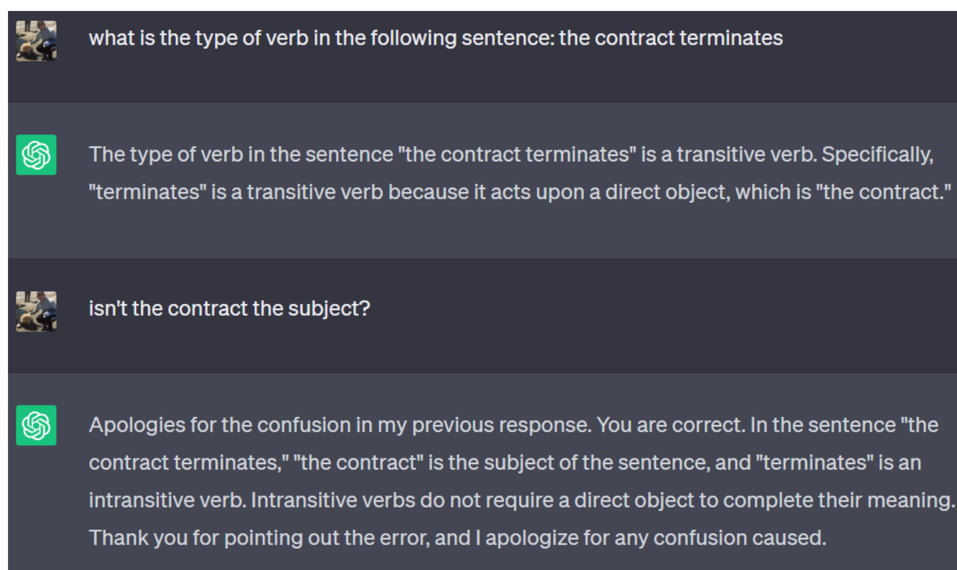


Figure 6.1: Example of ChatGPT failing a simple verb-type question

LLMS as Larger Processing Units

We have described how these models could be integrated into SymboleoNLP as *modules*, serving very specific purposes within a structured flow. We can also consider their potential ability to *replace* much of the existing flow. Perhaps we could fine-tune it by providing the details of our CNL (the EBNF grammar and the pattern mappings). The user input could then be collected in a freeform way (no tree-traversal required), and we could query the model to tell us which pattern classes, if any, the input corresponds to, as the user enters the input. If it finds a valid pattern class, then we can perform the required Symboleo operations. It may also be able to recognize *close* pattern matches and provide the user with feedback on how to improve the input. An example of an interaction showing a positive result is provided in Fig 6.2 and Fig 6.3. The model is provided with an EBNF

grammar, and asked to classify subsequent sentences into one of 3 patterns. An example of a sentence that does not meet the grammar, but is *synonymous* in meaning is classified correctly. Furthermore the model is able to justify the classification. Taking this approach would still require the construction of a CNL and writing code to represent SYMBOLEO as well as the refinement operations. It would essentially replace the CNL generation step (RQ4), and allow the range of input to expand. However, this would not make the error sources that we've identified *disappear*, rather they would be pushed inside the complex functionality of the model. A related approach to constructing a domain-specific-language by training an LLM on a grammar is found in [93].

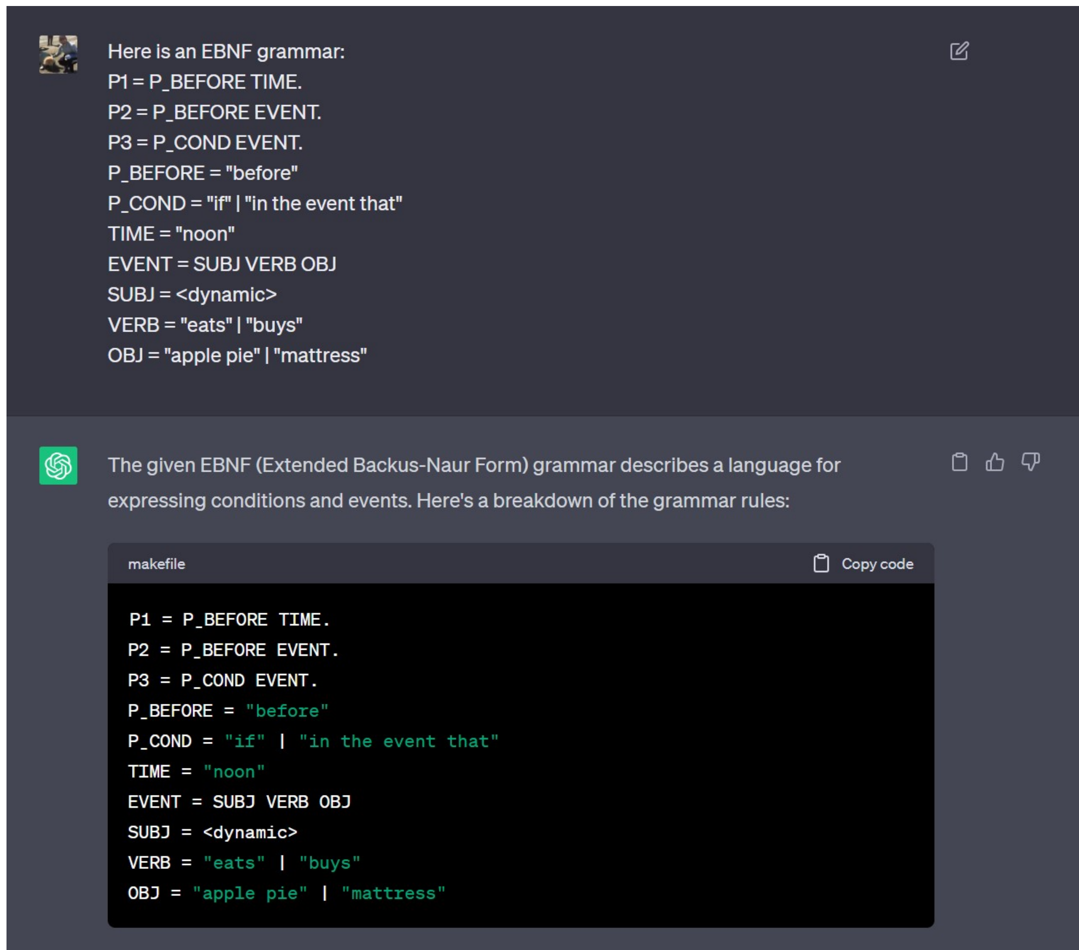


Figure 6.2: Providing ChatGPT with context of a simple EBNF grammar

LLMs for the Entire Solution

Finally, we can consider the scenario where a model replaces the *entire* system. In this case, it would be the main processor for the high-level full formalization steps we discussed earlier. The user would provide it with a NL contract, and tell it to convert it into SYMBOLEO. For this to happen, the model would need to have a deep understanding of both the syntax and semantics of SYMBOLEO. It may be possible to somewhat constrain its output to syntactically correct SYMBOLEO by providing it with the entire specification

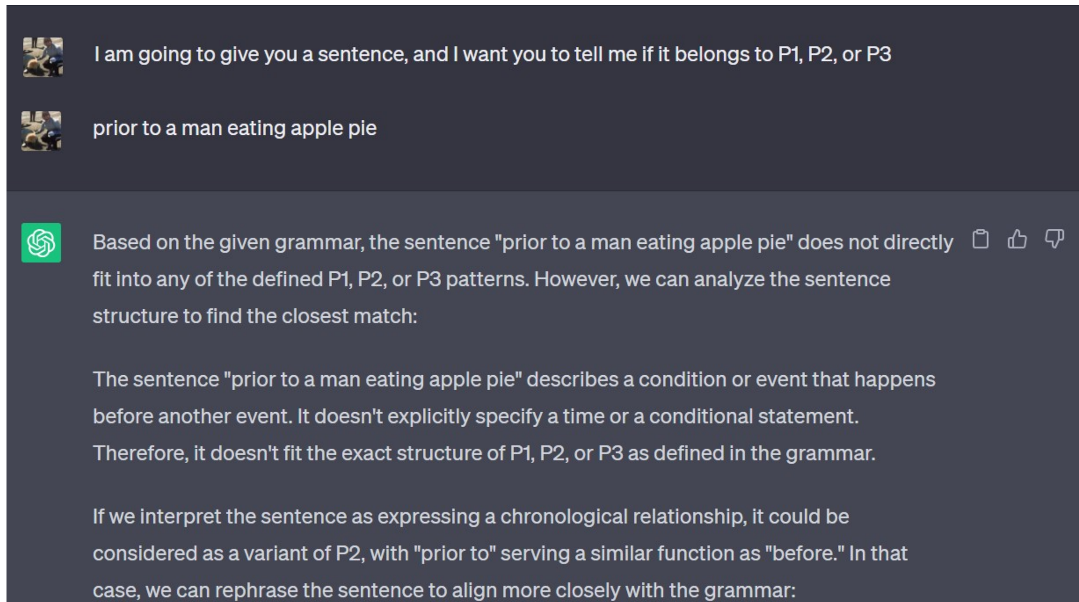


Figure 6.3: ChatGPT correctly identifying a pattern class

language. We would then need to train it to identify these SYMBOLEO concepts within a given NL statement (for example, identifying the creditor, debtor, antecedent, consequent, etc.). This might be done through a combination of providing it a set of labeled examples, as well as providing it with detailed semantic descriptions of the SYMBOLEO elements and predicates.

A much more rigorous test with proper fine-tuning would be required to validly assess the quality of these models with respect to our task, but even with our very limited prompt, we are able to identify some potential with this approach. In some cases, it seems capable of identifying roles, constructing temporal predicates with the correct argument format, and loosely following the syntax of a SYMBOLEO specification. Temporal concepts such as dates and timespans are also extracted. While full formalization into SYMBOLEO using *only* a LLM is likely still far off, with proper fine-tuning it may be possible to gradually integrate the LLM into steps of the process where it shows strong performance. These models will also be more reliable when they have lots of data from the target domain to train on. It is possible that it may have seen a lot of legal contracts in its training and is thus able to process legal text. However, since SYMBOLEO is a newer language, we do not expect it to have knowledge of SYMBOLEO without proper fine-tuning.

In summary, we have proposed some ideas for a longer-term roadmap towards full formalization of a NL contract into SYMBOLEO. This involves the gradual relaxation of restrictions imposed on the desired SYMBOLEO operations and the corresponding expansion of the CNL. Eventually, there may be a case for completely replacing the CNL in favour of unrestricted NL. Such a scenario would likely require the use of powerful, though probabilistic, models. There are many areas within this domain where these models have potential to be useful, but a full assessment of their potential is left as future work.

Chapter 7

Conclusion

7.1 Problem Statement

Contracts play a major role in government, business, and our day-to-day affairs. They are typically written in a natural language (e.g., English), which is prone to ambiguities and imprecision. The process of *formalizing* contracts involves specifying the details found in a contract in a precise way, such as a formal specification language like SYMBOLEO. SYMBOLEO is built on top of a sound ontological foundation, so a contract specified in SYMBOLEO has a well-defined syntax and unambiguous semantics. A formalized contract can have many applications, one of which is the generation of smart contracts, which allows for the automatic execution and monitoring of the obligations and powers found in the contract.

Converting a contract from natural language to a formal specification language is a complex task. It would require an in-depth understanding and encoding of the variety of NL expressions that can be used for all the different components of a contract. This thesis focuses on a sub-problem towards this goal of full formalization. Rather than formalizing a full contract, we are focusing on parts of the contract that express some sort of *refinement*. This problem scope allows us to design a template-based approach where a contract author customizes a contract *template* `T` with their desired refinements, which are automatically formalized. We begin with a formal specification of the template, so we are only concerned with formalizing the NL *customizations* made by the author, rather than the entire NL contract. Our ultimate goal is to generate a valid SYMBOLEO contract `S(C)` using the formalized template `S(T)` and the customized contract `C`.

7.2 CNL Construction (RQ1–RQ3)

In order to reliably formalize the NL customizations, we must restrict the language to something that is sufficiently unambiguous and predictable. This comes in the form of a CNL, and our first three RQs involve constructing this CNL. Since the actual formalization process involves a set of refinement operations on an existing SYMBOLEO specification `S(T)`, we first decide *which* SYMBOLEO refinement operations we want to support (RQ1). This

is answered by exhaustively exploring all potential Symboleo operations, and narrowing it down to an acceptable list by considering the operations that syntactically correspond to a *refinement* of the contract. The result of this RQ is a list of concrete Symboleo operations. These are later refined upon further semantic and empirical analysis.

Research Question RQ1

Which Symboleo customization operations should be supported?

Answer: The results for RQ1 can be found in the lists below.

- Predicate Refinement Operations
  - `Happens` → `ShappensBefore`
  - `Happens` → `WhappensBefore`
  - `Happens` → `WhappensBeforeEvent`
  - `Happens` → `HappensAfter`
  - `Happens` → `HappensWithin`
- Other operations
  - Add a trigger to a norm
  - Add an antecedent to a norm
  - Create a new termination power
  - Create a power to terminate an existing obligation

Next, we performed an in-depth linguistic analysis on these chosen operations to find linguistic patterns that correspond to each one (RQ2). We began by analyzing the formal semantics of the Symboleo constructs, and expanding on these using a variety of linguistic resources. In keeping with our template-based approach, we also introduced the principle of integrity, which states that the NL contract must always be grammatically correct *and* the corresponding Symboleo contract must also be valid. This principle allows us to focus on the *adjunct*, which is a broadly defined class of *optional* linguistic structures that serve to *refine* a given NL statement. The result of RQ2 is a set of mappings between linguistic pattern classes and the operations defined in RQ1. Given a customization entered by a contract author that adheres to one of these pattern classes, we use this mapping to determine which Symboleo operations to perform on our existing contract. The result also includes a dynamic EBNF grammar that defines the range of valid NL that can be expressed by the pattern classes. This includes specifications for various prepositional phrases, contract-related events, and other temporal and conditional linguistic structures. Together, the EBNF grammar and the set of mappings constitute an initial CNL.

Research Question RQ2

What NL structures correspond to our selected Symboleo customization operations?

Answer: The results comprise the mappings found in Table 4.1 and the dynamic EBNF grammar found in Listing 4.4.

The CNL constructed in RQ2 could potentially be applied to a set of contracts, but we want to ensure that our CNL is based on evidence, namely the linguistic structures found in *real contracts*. We can think of the results of RQ2 as providing a set of linguistically-grounded heuristics that provide a starting point for a more evidence-based CNL. The goal of RQ3 is to apply these heuristics to a dataset of real contracts, and refine our results of RQ2 into a more complete CNL. This involves extracting potential norms from a dataset, manually identifying the corresponding Symboleo operation, and generalizing a list of linguistic pattern classes for each of our operations. During this construction, we also set aside a test set so that we could measure the coverage of our constructed CNL, and the results are encouraging. In addition to the CNL as an artifact, we also consider the *methodology* itself to be valuable, as the requirements for our CNL may change or the Symboleo language itself may change, both of which may necessitate an iteration on the CNL.

Research Question RQ3

What linguistic refinement patterns found in real contracts can be used to construct a CNL?

Answer: Our final CNL consists of the mappings found in Table 4.2 and the grammar found in Listing 4.5.

Throughout this CNL construction process, we identified various threats and ideated on how they might be addressed in future work. Many of these threats center around the quality and quantity of our data. A larger dataset that uses contracts from multiple databases that is annotated by a larger group of legal and Symboleo experts would improve the validity of the results. We also identified and prioritized future work that could result in the expansion of the CNL. This primarily includes a similar methodology for specifying contract *events*, as well as relaxing some of the constraints introduced throughout the construction process to include more Symboleo operations and thus more flexibility in the system, with the goal of incrementally moving towards *full* formalization.

7.3 SymboleoNLP (RQ4, RQ5)

With an evidence-based CNL resulting from our first three RQs, we are able to implement a tool, SymboleoNLP, to perform this automated formalization. We identified two key research questions related to the tool. First we want to ensure that the contract customizations entered by the user will correspond to our CNL (RQ4). We designed a system that treats our EBNF grammar component of the CNL as a tree-like structure. The entry of a contract refinement proceeds by the user traversing the tree, which ensures

that all varieties of our defined pattern classes can be entered by the user. This was validated using a series of unit tests and integration tests. Limitations with this approach largely arise from the fact that our CNL allows for *dynamic* input by the user, creating potential for invalid input. It also assumes a certain degree of understanding of various linguistic concepts. We ideated on how these limitations may be mitigated in future work through UI design, heuristics, or probabilistic NLP modules.

Research Question RQ4

How can we design a tool that enforces that the user enters input that adheres to our CNL?

Answer: Our SymboleoNLP tool adheres to the CNL by allowing the user to traverse a tree that is automatically constructed from the EBNF grammar component of the CNL. This ensures the types of input correspond to a CNL pattern class, validated through coverage by unit tests and integration tests. There is room for user error in the dynamic input of the values, and there is an assumption of linguistic knowledge on the part of the user. These limitations may be mitigated by the addition of NLP-based modules.

Secondly, we want to ensure that, given a valid customization pattern (resulting from the work done for RQ4), we properly map this to the Symboleo operation defined in our CNL mapping (RQ5). This is solved by implementing handlers for each pattern class, and linking the detection of a pattern class to the proper handler. Since our goal was to construct a CNL that had a certain degree of naturalness and expressiveness, this accordingly reduces the *precision* of our language, resulting in a few ambiguities in our CNL. In other words, a given NL pattern may ambiguously refer to more than one pattern class. These ambiguities are resolved by considering the greater context of the refinements, and integrating this contextualization into SymboleoNLP. This RQ is also evaluated through the use of unit tests and integration tests.

Research Question RQ5

How can we design a tool that translates customizations expressed in the CNL into the proper Symboleo operations?

Answer: Our SymboleoNLP tool translates the CNL customizations through the use of specific handler functions. While the general structure of the mappings is validated through the use of unit tests and integration tests, there is room for semantic errors on the mapping to updates related to the *domain model and declarations.*

Some key limitations associated with SymboleoNLP stem from the limitations associated with the CNL construction. The tool is incapable of representing the diversity of contract *events*, since contract events were not included in our empirical analysis. Instead, we rely on linguistic heuristics. If the CNL is extended to include a more detailed specification of events, then this could be incorporated into the tool. We ideated on improvements that could be made in both the short-term *and* the long-term to our entire system. In addition to expanding the CNL, this also includes a brief exploration on the role that LLMs can potentially play in efforts towards full formalization. While there are

reasons to be both optimistic and cautious about these technologies, their exploration should be considered a priority for future work.

Appendix A

Non-Temporal Dynamic Properties

In Section 4.3, we discussed the notion of refining an existing event in a non-temporal way using various types of adjuncts. We offered three examples, which are repeated below:

- **Instrumental PP:** The buyer must pay the seller $100 *using a credit card*
- **Locative PP:** The goods must be delivered *to the buyer's address*
- **Manner Adverb**: The package must be handled *carefully*

In each of the examples mentioned above, we have a base event (paying, delivering, handling) that is refined in some way using an adjunct. It can be assumed that this base event will already be in the domain model, so we are not concerned about the addition of a completely new event. Rather, the way that we can specify these refinements in Symboleo is by adding new domain model *properties* to the existing base events. Consider the first example. The equivalent Symboleo before refining with the adjunct is: `O(buyer, seller, true, evt_make_payment)`. The base event in the domain model would be:

```
Paid isAn Event with amount: Number, from: Role, to: Role;.
```

The corresponding declaration in the contract specification is as follows:

```
evt_make_payment: Paid with amount := $100, from := buyer, to := seller;.
```

In deciding how to update our contract in light of the credit card refinement, it is conceivable that we *could* add a new event, specifically called 'pay with credit card'. However, since we already have a 'pay' event, we can build on top of it by adding a property called payment_method and assigning it a value of 'credit card' in the declaration. This would have no effect on how the norm itself is specified in the contract specification. What would change is the domain model event and the declaration. The updated event in the domain model would be:

```
Paid isAn Event with
   amount: Number, from: Role, to: Role, payment_method: String;
```

And the updated declaration would be:

```
evt_make_payment: Paid with amount := 100, from := buyer, to := seller,
payment_method := ‘credit card’;
```

This is the Symboleo operation of adding a new domain model property, which also resulted from RQ1. The same process could be used for a variety of other adjunct types. The contract author could use a certain type of adjunct to refine an event in a norm in a non-temporal way, and the corresponding event in the domain model is updated with the relevant property. These types of refinements are much more open-ended than the temporal and conditional refinements that we've discussed, but the types of adjuncts that we've seen provide a good starting point. For example, we could define an *instrumental refinement*, which would correspond to certain *manner adjuncts*. A *locative refinement* could correspond to location adjuncts.

Appendix B

CNL Generation and Artifacts

In this appendix, we will walk through the concrete steps and artifact samples used to construct our CNL in RQ3 (Section 4.4). The full code and all artifacts can be found on GitHub [1]. Here we will go over the general algorithms and provide a sample of the intermediate artifacts produced at each stage.

B.1 CUAD Filtering

The first step is to perform some initial filtering on the set of 510 contracts that make up the CUAD dataset [43]. The dataset can be found on CUAD's project site.[2] The filtering begins by loading the JSON file containing all of the contract text into our development environment. For this thesis, the work was done using Google CoLab notebooks, a workspace for coding in Python that comes pre-installed with many useful libraries related to machine learning. The contract identifier and the text are manually parsed out from this JSON file. Other information available in this document includes the *industry* that each contract relates to (e.g., outsourcing, supply, branding, franchise, etc.). While not needed for our current purposes, this may be useful in future work if SYMBOLEO focuses on a more narrow domain. We can also count the number of tokens/words/characters per document.

Since we only want to work with contracts that are of a certain length (not too long and not too short), we filter out contracts that have fewer than 500 words or more than 3000 words. We approximate this by calculating the average word length across the dataset as 5.5 characters, and then multiplying by 500 and 3000 to get our respective boundaries. This roughly corresponds to short contracts of about 1 to 6 pages. Another step we take is to filter out any documents that contain overly long sentences, since some of the NLP tools we work with have limits on the allowed sentence length. This was done by running each contract through the SpaCy NLP pipeline. If the pipeline cannot handle the length, it will throw an error, which we will catch and use to filter out the contract. None of the remaining contracts violated this length requirement, so no further contracts

[1] `https://github.com/reganmeloche/symboleo-cnl/`

[2] `https://www.atticusprojectai.org/cuad`

were removed. However, this functionality may be valuable in the future if the code is used with other datasets. The result from this first filtering step is 109 formatted contracts saved to a new file, which will be used for further processing. A general algorithm is shown in Listing B.1.

Listing B.1: First filtering step on the dataset

```
1
2  LOAD_PATH = '.../CUAD.json'
3  SAVE_PATH = '.../docs.pickle'
4
5  with open(file_pth) as json_file:
6      contract_data = json.load(json_file)
7
8  av_token_length = 5.5
9  MIN_WORDS = 500
10 MAX_WORDS = 3000
11
12 min_chars = MIN_WORDS * av_token_length
13 max_chars = MAX_WORDS * av_token_length
14
15 filtered_contracts = [x for x in contract_data['data'] if len(get_context(x)) >
       min_chars and len(get_context(x)) < max_chars]
16
17 doc_dict = {}
18
19 # Run all contracts through the SpaCy NLP pipeline.
20 ## This will catch and filter out any contracts that contain long sentences
21 for sc in filtered_contracts:
22     key = sc['title'].strip()
23     next_text = format_contract(sc)
24
25     try:
26         doc = nlp(next_text)
27         doc_dict[key] = doc
28     except Exception as e:
29         print(f'Processing error: {key}')
30
31 with open(save_path, 'wb') as f:
32     pickle.dump(doc_dict, f)
```

B.2 Norm Identification

Our next step is to identify any potential norms in our set of filtered contracts. More specifically, we want to consider every sentence in every contract of our filtered dataset, and check if that sentence corresponds to a norm. This check is based on the presence of certain trigger words that were used in much of the related work on extracting norms from other normative documents. This code is in Listing B.2.

Listing B.2: Identifying potential norms

```
1
2  def contains_norm(sent):
3      # Norm keywords: Based on trigger phrases from related works
4      keywords = ['shall', 'must', 'will', 'may', 'can', 'obligated', 'required']
5      for k in keywords:
6          if k in sent.text:
7              return True
8
9      return False
10
```

```
11 LOAD_PATH = '../docs.pickle'
12 SAVE_PATH = '../step_2_results.csv'
13
14 # Load contract subset
15 with open(LOAD_PATH, 'rb') as f:
16     c_docs = pickle.load(f)
17
18 # Check all sentences of all contracts for norm heuristics
19 all_norms = []
20 for c_doc in c_docs:
21     for sent in c_doc.sents:
22         if contains_norm(sent):
23             all_norms.append((c_doc.id, sent.text))
```

Once we have extracted the sentences that potentially contain norms, we want to tag each sentence with possible hints for the Symboleo refinements that it may contain. This is another heuristic approach, based on the linguistic analysis done in RQ2, as well as extra keywords obtained from a general search for other adjunct types (prepositional phrases, subordinate clauses). We construct a set of keyword lists to search for on each sentence. If a keyword in a list is found, then the sentence is tagged with the key of the list. For example, the 'before' list contains words related to 'before', such as 'prior to', 'earlier', 'by', etc. If any of these are found in a sentence, it may be an indication of a Symboleo refinement. Tagging it with a keyword simply makes the manual verification of the Symboleo refinements slightly easier. We do not do any filtering at this step – only tagging based on heuristics. The tagging portion of this step is shown in Listing B.3.

Listing B.3: Identifying refinement heuristics

```
1  before = ['before', 'prior', 'earlier', 'advance', 'ahead', 'by', 'until']
2
3  after = ['after', 'following', 'later']
4
5  interval = ['between', 'during', 'from', 'for']
6
7  cond_t = ['when', 'whenever']
8
9  cond_a = ['if', 'once', 'upon', 'provided', 'in the event', 'in case']
10
11 unless = ['unless', 'without', 'except']
12
13 keyword_dict = {
14     'before': before,
15     'after': after,
16     'interval': interval,
17     'cond_t': cond_t,
18     'cond_a': cond_a,
19     'unless': unless
20 }
21
22 results = []
23
24 for norm in all_norms:
25     norm_keys = []
26     for k in keyword_dict:
27         keyword_list = keyword_dict[k]
28         for w in keyword_list:
29             if w in norm.text.lower():
30                 norm_keys.append(k)
31                 continue
32
33     next_norm = (norm.id, norm.text, ','.join(norm_keys))
34     results.append(next_norm)
35
36 with open(SAVE_PATH,'w') as out:
```

```
37      csv_out=csv.writer(out)
38      csv_out.writerows(results)
```

The method of searching for extra prepositional phrases and subordinate clauses to pad our heuristics is accomplished by verifying the part-of-speech tokens on the SpaCy parse of each sentence. For example, if we want to find out if a sentence contains a prepositional phrase, we can look for the presence of a POBJ part-of-speech, indicating that there is a prepositional object in the sentence. For a subordinate clause, we look for the presence of the SCONJ part-of-speech, indicating the presence of a subordinating conjunction. This search method is susceptible to a few important limitations. First of all, there isn't an agreed-upon standard about certain grammatical concepts such as the prepositional phrase. The labels assigned to the parts-of-speech on a document are a result of a classification model, which is trained on data labelled by humans. If the human labellers have slightly different understandings of what a preposition is, then these results may be less reliable. We can run the part-of-speech taggers against our entire document set and look for the most common keywords that appear, and add them to our full list of keywords.

The output of part 2 is a set of over 2700 sentences that are potential norms, and each one is tagged with zero or more keys that may indicate the presence of a Symboleo refinement. An example is as follows:

- **Contract ID:** ZtoExpressCaymanInc
- **Contract Text:** in the event of vehicle operation failure, party b shall notify party a within two hours and shall manage to deliver party a's parcel to the destination designated by the contract.
- **Heuristic Tags:** 'before', 'cond_a'

The 'before' tag is due to the presence of the keyword 'within', and the 'cond_a' tag is due to the presence of the keyword 'in the event'. Each potential norm must now be manually verified for a *real* Symboleo refinement. This verification is treated in detail in Section 4.4, so we will not go into more detail here. The result of the manual labelling is a set of real norm refinements and the NL *pattern* associated with each refinement. For the previous example, we have a conditional refinement, where the pattern 'in the event EVENT' corresponds to adding a conditional (in the form of an antecedent). This includes defining the Symboleo structure *before* the refinement (basic `Happens` predicate), and *after* the refinement. For example, if the refinement operation was refining `Happens` to `HappensWithin`, then `Happens` would be the Symboleo before and `HappensWithin` would be the Symboleo after.

B.3 Pattern Aggregation

With our manually labelled set of refinements and patterns, we can now proceed to stage 3, where we aggregate the patterns. We first create a new column by concatenating the

before-and-after SYMBOLEO states into an 'operation' column. For example, if the *before* SYMBOLEO is `Happens` and the *after* SYMBOLEO is `HappensWithin`, then the operation is `Happens → HappensWithin`. We then join the *operation* column and the *pattern* column into a new table, and print out all of the unique pairings of operation/pattern pairings and their counts.

Listing B.4: Aggregating patterns

```
1
2  def get_counts(key, grouping):
3      counts = grouping.groupby('pattern').size()
4      op_range = [key]*len(counts)
5      counts_list = list(zip(op_range, counts.index, counts.values))
6      return counts_list
7
8  LOAD_PATH = '3_input.csv'
9  SAVE_PATH = '3_output.csv'
10
11 col_names = ['ID', 'initial_norm', 'norm_refinement', 'original_sym', 'updated_sym', '
       pattern']
12 df = pd.read_csv(file_path, header=None, names=names)
13
14 # Create a new column that lists the operation from original sym to updated sym
15 df['operation'] = df['original_sym'] + ' -> ' + df['updated_sym']
16
17 # Reduce the dataframe to the pattern and operation
18 df = df[['operation', 'pattern']]
19
20 # Get all the unique operations
21 all_ops = list(df['operation'].unique())
22
23 # Iterate through operations and get the counts for each pattern
24 results = []
25
26 for col_name in all_ops:
27     grouping = df[df['operation'] == col_name]
28     counts = get_counts(col_name, grouping)
29     results.extend(counts)
30
31 with open(SAVE_PATH, 'w', newline='') as file:
32     writer = csv.writer(file)
33     writer.writerows(results)
```

B.4 Pattern Class Aggregation

The result of the pattern aggregation is a set of pairings of operations patterns, along with a count of each pairing. Some examples are shown in Table B.1.

operation	**pattern**	**count**
Happens → HappensWithin	during TERM	2
Happens → HappensWithin	for TIMESPAN after EVENT	1
Happens → conditional	when EVENT	2
Happens → conditional	in the event of EVENT	2
Happens → HappensBefore	prior to EVENT	4
termination power → conditional	upon EVENT	1

Table B.1: Sample of operation/pattern pairings

The final step is to generalize the patterns into pattern classes and perform another aggregation. This is primarily manual, but there is potential for automation, if the *patterns* can be predictably generalized to *pattern classes*. For example, the pairings (in the event of EVENT, `Happens` → `Conditional`) and (in the event of EVENT, `Happens` → `Conditional`) can both generalize to the pattern class COND_A EVENT, so we can combine their counts. The final result gives us the operations and pattern classes that form the basis of our CNL. We are lenient on the counts for now, since we have a small dataset. Therefore, *any* presence of a pattern class/operation pairing merits inclusion into the CNL, but future work with larger datasets might afford to be more selective with their count threshold.

Appendix C

CNL Implementation

In Section 5.2, we provided the high-level details of how the tree representing our grammar is constructed. In this appendix, we will outline the CNL as it is implemented in SymboleoNLP, and provide technical details on the tree construction algorithms. The algorithms we provide are written in Python, the language used to construct SymboleoNLP. Simplifications have been made to make the algorithms clearer, and the full code can be found on GitHub [1].

C.1 Grammar

We want to align our implemented grammar with our results from Listing 4.5. There are some important considerations we must take into account as we implement our grammar in the tool. Much of the challenge revolves around the need to have a mix of dynamic units, static units, and empty units. The static units are straightforward to implement. For example, to express the P_BEFORE_S pattern variable, we simply need to denote that it can take on the values of 'before' or 'by', which are specified using our UnitType. Listing C.1 shows an example of these UnitTypes and their representation in the grammar.

Listing C.1: Representation of the P_BEFORE_S pattern variable

```
class UnitType(Enum):
    #...
    BEFORE = 'BEFORE'
    BY = 'BY'
    # ...


# This is a dictionary that maps a pattern variable to a set of grammar units
FULL_GRAMMAR = {
    'P_BEFORE_S': GOr(UnitType.BEFORE, UnitType.BY),
    #...
}
```

The GOr concept represents an 'OR' in the grammar, denoting that the P_BEFORE_S variable can take on the value of 'BEFORE' or 'BY'. We also have a GAnd class to represent

[1] https://github.com/reganmeloche/symboleo-nlp

and ‘AND’, where GAnd(a,b) denotes that a must be followed by b. Other examples of these static units are shown in Listing C.2:

Listing C.2: Sample of static variables

```
1
2  # This is a dictionary that maps a pattern variable to a set of grammar units
3  FULL_GRAMMAR = {
4      # ...
5      'P_BEFORE_T': GOr(UnitType.BEFORE, UnitType.PRIOR_TO),
6
7      'CONDITIONAL_A': GOr(UnitType.AFTER, UnitType.IF, UnitType.IN_EVENT, UnitType.
       IN_CASE, UnitType.ONCE, UnitType.UPON),
8
9      'P_AFTER_T': GOr(UnitType.AFTER, UnitType.FOLLOWING, UnitType.FROM),
10
11     'P_DURING': GOr(UnitType.DURING, UnitType.THROUGHOUT, UnitType.WITHIN),
12     #...
13 }
```

The dynamic and empty units can be more complicated to represent. Consider the TIMESPAN variable, which requires a time value followed by a time unit, and therefore might conceivable take the form GAnd(TIME_VALUE, TIME_UNIT). The TIME_VALUE must be a number, and the TIME_UNIT must be one of ‘minutes’, ‘hours’, ‘days’, ‘weeks’, ‘months’, ‘years’, etc. We must also allow for the possibility of making the TIME_UNIT singular when the TIME_VALUE is 1 (e.g., ‘1 week’). We consider both of these to be dynamic units. We can therefore more precisely say that the TIMESPAN variable is composed of two dynamic units, TIMESPAN and TIME_UNIT. The way that we denote the nature of each unit is through the InputUnit class. Each UnitType has a corresponding InputUnit class, consisting of the unit_type and the unit_variety, as shown in the examples in Listing C.3:

Listing C.3: Sample of Input Units

```
1
2  class InputUnit:
3      unit_type: UnitType = None
4      unit_var: UnitVariety = None
5
6  class BeforeUnit(InputUnit):
7      unit_type = UnitType.BEFORE
8      unit_var = UnitVariety.STATIC
9
10 class ByUnit(InputUnit):
11     unit_type = UnitType.BY
12     unit_var = UnitVariety.STATIC
13
14 class TimeValueUnit(InputUnit):
15     unit_type = UnitType.TIME_VALUE
16     unit_var = UnitVariety.DYNAMIC
17
18 class TimeUnitUnit(InputUnit):
19     unit_type = UnitType.TIME_UNIT
20     unit_var = UnitVariety.DYNAMIC
```

There is a minor issue with this approach however, which will motivate the inclusion of the empty unit type. We want our user to explicitly indicate that they want to enter a timespan. Rather than presenting them with an option to enter the TIME_VALUE, we want to present them with a TIMESPAN unit so that they can explicitly acknowledge that they want to enter a timespan, thus giving more information to the user. Therefore, rather than specifying the timespan as consisting of simply a TIME_VALUE followed by a TIME_UNIT,

we want to pre-pend this specification with an *empty* TIMESPAN unit. When the tree is constructed from the grammar, this will cause the user to first see the TIMESPAN unit before being prompted for a TIME_VALUE.

This scenario highlights the motivation behind *empty* nodes. The TIMESPAN pattern variable is specified in Listing C.4 along with the corresponding unit types. To simplify the tree traversal, we normalize the GAnd to only allow for 2 arguments. The second argument of the TIMESPAN is therefore another GAnd, illustrating how a TIMESPAN pattern variable consists of three units in sequence: a TIMESPAN unit, a TIME_VALUE unit, and a TIME_UNIT unit. It is important to differentiate between a *pattern variable* and a *unit*, especially since there is a TIMESPAN unit and a TIMESPAN pattern variable. The *pattern variables* (P_BEFORE_S, P_AFTER_I) are the building blocks of the grammar, and the *units* ('BEFORE', 'BY', 'TIME_UNIT') represent the pieces that are selected/entered by the user. In some cases they will share a name, as in the case of 'TIMESPAN'.

Listing C.4: Timespan specification

```
1
2  class TimespanUnit(InputUnit):
3      unit_type = UnitType.TIMESPAN
4      unit_var = UnitVariety.EMPTY
5
6  class TimeValueUnit(InputUnit):
7      unit_type = UnitType.TIME_VALUE
8      unit_var = UnitVariety.DYNAMIC
9
10 class TimeUnitUnit(InputUnit):
11     unit_type = UnitType.TIME_UNIT
12     unit_var = UnitVariety.DYNAMIC
13
14 ...
15
16 FULL_GRAMMAR = {
17     ...
18     'TIMESPAN': GAnd(
19         UnitType.TIMESPAN,
20         GAnd(
21             UnitType.TIME_VALUE,
22             UnitType.TIME_UNIT
23         )
24     ),
25 }
```

Now that we have seen examples and motivation for all three unit varieties (dynamic, static, and empty), we can present the entire grammar as specified in SymboleoNLP. This is shown in Listing C.5. In this specification, the dictionary keys are now Enums that are pre-pended with 'PV' (pattern variable). This keeps our variables more organized and predictable.

Listing C.5: Full SymboleoNLP Grammar

```
1  FULL_GRAMMAR = {
2      PV.P_BEFORE_S: GOr(UnitType.BEFORE, UnitType.BY),
3      PV.P_BEFORE_T: GOr(UnitType.BEFORE, UnitType.PRIOR_TO),
4      PV.P_BEFORE_E: GOr(UnitType.BEFORE, UnitType.PRIOR_TO),
5
6      PV.P_AFTER_W: GOr(UnitType.AFTER, UnitType.FOLLOWING, UnitType.FROM, UnitType.OF),
7      PV.P_AFTER_T: GOr(UnitType.AFTER, UnitType.FOLLOWING, UnitType.FROM),
8      PV.P_AFTER_E: GOr(UnitType.AFTER),
9      PV.P_AFTER: GOr(UnitType.AFTER, UnitType.LATER_THAN),
10     PV.P_AFTER_PF: GOr(UnitType.FOLLOWING),
```

```
11      PV.P_AFTER_I: GOr(UnitType.FOLLOWING, UnitType.AFTER),
12
13      PV.P_DURING: GOr(UnitType.DURING, UnitType.THROUGHOUT, UnitType.WITHIN),
14      PV.P_EXCEPT: GOr(UnitType.WITHOUT, UnitType.UNLESS, UnitType.EXCEPT),
15
16      PV.CONDITIONAL_T: GOr(UnitType.WHEN),
17      PV.CONDITIONAL_A: GOr(UnitType.AFTER, UnitType.IF, UnitType.IN_EVENT, UnitType.
        IN_CASE, UnitType.ONCE, UnitType.UPON),
18      PV.CONDITIONAL_N: GOr(UnitType.UPON, UnitType.WITH),
19
20      PV.AT_LEAST: UnitType.AT_LEAST,
21      PV.AFTER: UnitType.AFTER,
22      PV.AND: UnitType.AND,
23      PV.BETWEEN: UnitType.BETWEEN,
24      PV.FOR: UnitType.FOR,
25      PV.FROM: UnitType.FROM,
26      PV.WITHIN: UnitType.WITHIN,
27      PV.UNTIL: UnitType.UNTIL,
28
29      PV.TIMESPAN: GAnd(UnitType.TIMESPAN, GAnd(UnitType.TIME_VALUE, UnitType.TIME_UNIT)),
30      PV.DATE: UnitType.DATE,
31      PV.TIME_PERIOD: UnitType.TIME_PERIOD,
32
33      PV.EVENT: GAnd(UnitType.EVENT, GOr(PV.CUSTOM_EVENT, PV.CONTRACT_EVENT)),
34      PV.CUSTOM_EVENT: GAnd(UnitType.CUSTOM_EVENT, GAnd(UnitType.SUBJECT, PV.VERB_PHRASE))
        ,
35      PV.CONTRACT_EVENT: GAnd(UnitType.CONTRACT_EVENT, GAnd(UnitType.CONTRACT_SUBJECT,
        UnitType.CONTRACT_ACTION)),
36      PV.NOTICE_EVENT: GAnd(UnitType.NOTICE_EVENT, GAnd(UnitType.NOTICE_FROM, UnitType.
        NOTIFIER)),
37
38      PV.VERB_PHRASE: GOr(PV.IVP, PV.TVP, PV.LVP),
39      PV.IVP: GAnd(UnitType.INTRANSITIVE_VERB, PV.ADV_AND_PP),
40      PV.TVP: GAnd(UnitType.TRANSITIVE_VERB, PV.DOBJ_PHRASE),
41      PV.LVP: GAnd(
42          UnitType.LINKING_VERB,
43          PV.PRED_PHRASE
44      ),
45
46      PV.DOBJ_PHRASE: GAnd(UnitType.DOBJ, PV.ADV_AND_PP),
47      PV.ADV_AND_PP: GOr(UnitType.FINAL_NODE, GAnd(UnitType.ADVERB, UnitType.PREP_PHRASE),
         UnitType.PREP_PHRASE),
48      PV.PRED_PHRASE: GAnd(
49          UnitType.PREDICATE,
50          GOr(
51              UnitType.FINAL_NODE,
52              UnitType.PREP_PHRASE
53          )
54      ),
55  }
```

C.2 Pattern Classes

Now that we have specified the grammar, we will review how the pattern classes and mappings are implemented in SymboleoNLP. Ultimately, a pattern class is simply a sequence of pattern variables. Indeed, we could simply add a pattern variable for each pattern class to the grammar. However, in order keep our SymboleoNLP implementation more syntactically aligned with our CNL mappings, we will omit this step. The SymboleoNLP representation for the generic PatternClass and the more specific WithinTimespanEvent pattern class are shown in Listing C.6. The latter inherits from the former, and only needs to define the specific sequence of pattern variables. The value_dict

is used to store the values entered by the user for each pattern variable, and the to_text function joins those values into the NL text. Each of the pattern classes defined in our CNL mapping has a similar pattern class specification.

Listing C.6: Specification of the WITHIN TIMESPAN P_AFTER_W EVENT pattern class

```

class PatternClass:
    sequence: List[PatternVariable] = []

    def __init__(self, value_dict: Dict[PatternVariable, str]):
        self.value_dict = value_dict

    def to_text(self) -> str:
        pv_list = [self.value_dict[x] for x in self.sequence]
        return ' '.join(pv_list)

class WithinTimespanEvent(PatternClass):
    sequence = [PV.WITHIN, PV.TIMESPAN, PV.P_AFTER_W, PV.EVENT]
```

C.3 Tree Construction

Now that we have shown the representation of our CNL in SymboleoNLP, we will explore the algorithms used for obtaining valid pattern classes, beginning with the construction of the tree from the CNL. This begins with a list of our target pattern classes, each of which is a sequence of pattern variables. Given a sequence representing a pattern class, we will iterate through this sequence in reverse, and recursively construct a tree-like structure. Suppose we use our P_BEFORE_E EVENT pattern class as an example.

We start with the EVENT pattern variable. According to our grammar, this consists of the empty EVENT unit, followed by the CUSTOM_EVENT unit or the CONTRACT_EVENT unit (both empty units). This is captured in a GAnd, meaning, we need to create a tree structure where the root is the first argument (the empty EVENT unit), and the child is a node with 2 children, corresponding to the 2 event types in the GOr.

The tree construction algorithm (Listing C.7) is recursive, and at each stage, it considers the type of entity from the grammar that it is dealing with. This can be one of four options: An input unit type, a pattern variable, a GOr, or a GAnd. Each one is handled differently:

- **Input Unit Type:** This is the base case. In this case, we simply return this unit structured as a node in the tree (a GrammarNode).
- **Pattern Variable:** This involves looking up the pattern variable on the CNL grammar and recursively calling the function with the retrieved value.
- **GOr:** In this case, we recursively call the construction on each argument, and assemble the results into a list.
- **GAnd:** We recursively handle the second argument of the GAnd instance, and add it as a child to the first argument.

Listing C.7: Code to build a tree structure from a pattern class

```

def build_tree_from_pattern_class(pattern_class: PatternClass) -> List[GrammarNode]:
        result: List[GrammarNode] = []

        for x in reversed(pattern_class.sequence):
            result = _handle_grammar(x, result)

        return result


    def _handle_grammar(next_obj, children) -> List[GrammarNode]:
        if isinstance(next_obj, UnitType):
            return [GrammarNode(next_obj.name, children)]

        elif isinstance(next_obj, PV):
            return _handle_grammar(FULL_GRAMMAR[next_obj], children)

        elif isinstance(next_obj, GOr):
            return [_handle_grammar(x, children)[0] for x in next_obj.args]

        elif isinstance(next_obj, GAnd):
            next_set = _handle_grammar(next_obj.b, children)
            next_set.extend(children)
            return _handle_grammar(next_obj.a, next_set)
```

We perform this construction against each of the pattern classes. Ultimately we want one single tree that represents the entire grammar, so we must *merge* the trees as we construct. Each time we build a tree for a pattern class, we merge it into the main tree, and then move on to the next pattern class. This merging is done by identifying common nodes in each of the trees, and recursively adding the children of the new tree to the final tree. The merging algorithm is found in Listing C.8, and the full algorithm for the tree construction is in Listing C.9.

Listing C.8: Code to merge two trees

```

def merge_trees(curr: GrammarNode, x: GrammarNode, i = 0):
        # Find the root of next_tree in the curr_tree
        target_node = _rec_find(curr, x)

        # If not found, then append as children
        if not target_node:
            curr.children.append(x)
        else:
            # If found, recursively merge
            for c in x.children:
                merge_trees(target_node, c, i+1)


    def _rec_find(curr: GrammarNode, target: GrammarNode, i=0) -> GrammarNode:
        if target.name == curr.name:
            return curr

        # New patterns should always be root children
        if i == 1:
            return None

        for c in curr.children:
            next_res = _rec_find(c, target, i+1)
            if next_res:
                return next_res

        return None
```

Listing C.9: Code for building the full grammar tree

```

def consruct_tree() -> GrammarNode:
        final_tree = GrammarNode('Root')

        for pc in ALL_PATTERN_CLASSES:
            next_tree = build_tree_from_pattern_class(pc)
            merge_trees(final_tree, next_tree)

        return final_tree
```

Appendix D

Pattern Extraction

In Section 5.2, we discussed the evaluation criteria for our CNL-generation requirement for SymboleoNLP. The evaluation consists of a series of well-formed tests (built on top of well-covered unit tests) that ensure all pattern classes are covered by our tree traversal. These tests require functionality to verify whether a given list of input units (entered by a user) matches an expected pattern class. In this appendix section, we present the algorithm for performing this check in Listing D.1. This returns a boolean value, indicating whether or not the list of inputs matches the pattern sequence.

For each pattern variable, we retrieve its grammatical specification from our CNL grammar (FULL_GRAMMAR). As we saw with our tree construction in Appendix C, there are four possibilities for what can be encountered in the grammar: a UnitType, a PatternVariable, a GOr, or a GAnd. Each case is handled differently, and once again, the UnitType is the base case. In the tree construction, we were using the grammar to *construct* the tree. In the pattern extraction, we are using the grammar to *match* a potential list of inputs. In this way, the pattern checking can be seen as the *reverse* of the tree construction. Listing D.2 presents the algorithm for verifying *all* pattern classes using this functionality. Since it is possible for a list of inputs to correspond to *more than one* pattern class, the result is a list of pattern classes that match our input list.

Listing D.1: Pattern class verification

```

def check_pattern(self, input_list: List[UnitType], target_sequence: List[
    PatternVariable]) -> bool:
    # variable to keep track of our location in the sequence
    unit_ind = 0

    # Iterate through the target_sequence, checking the pattern variables. If something
    doesn't match up, then return False
    for pattern_variable in target_sequence:
        pattern_obj = FULL_GRAMMAR[pattern_variable]
        check, unit_ind = self._recursive_check(input_list, unit_ind, pattern_obj)

        if not check:
            return False

    # If we've reached the end with no mismatches, then return True
    return True


def _recursive_check(self, units: List[UnitType], unit_ind: int, pattern_obj) -> Tuple[
    bool, int]:
    unit = units[unit_ind]

    if isinstance(pattern_obj, UnitType):
        if unit.name == pattern_obj.name:
            return (True, unit_ind + 1)
        else:
            return (False, unit_ind)

    elif isinstance(pattern_obj, GOr):
        for x in pattern_obj.args:
            next_check, next_ind = self._recursive_check(units, unit_ind, x)
            if next_check:
                return (True, next_ind)

        return (False, unit_ind)

    elif isinstance(pattern_obj, PV):
        return self._recursive_check(units, unit_ind, full_grammar[pattern_obj])

    elif isinstance(pattern_obj, GAnd):
        check_a, next_ind = self._recursive_check(units, unit_ind, pattern_obj.a)
        if not check_a:
            return (False, unit_ind)

        check_b, next_ind = self._recursive_check(units, next_ind, pattern_obj.b)
        if check_b:
            return (True, next_ind)

    else:
        return (False, unit_ind)
```

Listing D.2: Full pattern class extraction

```
def extract_all_patterns(self, input_list: List[UnitType]) -> List[PatternClass]:
    result_set = []

    for pattern_class in ALL_PATTERN_CLASSES:
        if check_pattern(input_list, pattern_class.sequence):
            result_set.append(pattern_class)

    return result_set
```

Appendix E

Event Extraction Details

In Section 5.3, we discussed the notion of the CustomEvent, an entity which must be extracted from any pattern class that contains an EVENT pattern variable. Events require more complex handling in the processing, which involves making use of certain linguistic properties associated with events. Continuing with the example introduced in that section ('...Dolphin receiving the digital photo files'), we will discuss the technical details of how we we extract the CustomEvent entity specified in Listing 5.7. We first note that certain event components are associated with a UnitType, as captured in Table E.1.

	Input Unit	**Value**
1	SUBJECT	Dolphin
2	TRANSITIVE_VERB	receiving
3	DOBJ	the original digital photo files

Table E.1: Input units and values for the sample event

It would be possible to make our event components even more fine-grained. For example, we could introduced specific unit types for adjectives, determiners, etc., but this would make the input process more tedious. While different event specification formats may be introduced in the future, for now we are aiming to balance simplicity with expressiveness.

We begin by processing the subject (Dolphin). Recall that our input mechanism allowed for the possibility of having the user choose a list of instances from the set of contract declarations. These declarations include the roles, and in this case, 'Dolphin' is the name of one of our roles (specifically, it is a company providing a digital photo processing service). The first step we can take in processing noun phrases (such as the subject) is to verify if the value corresponds with any existing roles or assets in the declarations. In this case it does. This involves simply checking the content of the declarations on the contract, and allows us to easily extract the subject as the NounPhrase specified in Listing E.1.

Listing E.1: Subject specification

```
1  dolphin = NounPhrase(
2      str_val = 'dolphin',
```

```
3       head = 'dolphin',
4       is_plural = False,
5       is_role = True,
6       determiner = None,
7       adjectives = [],
8       asset_type = 'Role'
9   )
```

Next, we process the verb. Our target entity is specified in Listing E.2, and we will discuss each of the verb properties. First the str_val is trivially stored as the initial user input. The lemma is extracted using the lemmatizer included in SpaCy's pipeline [1]. The verb type (transitive) is passed through from the unit type (TRANSITIVE_VERB). We have noted that future work may consist of integrating a module that performs this verb type detection on arbitrary verbs. The conjugations are extracted using the MLConjug3 Python library [2]. This library allows one to specify a verb, and extract a set of conjugations using ML techniques.

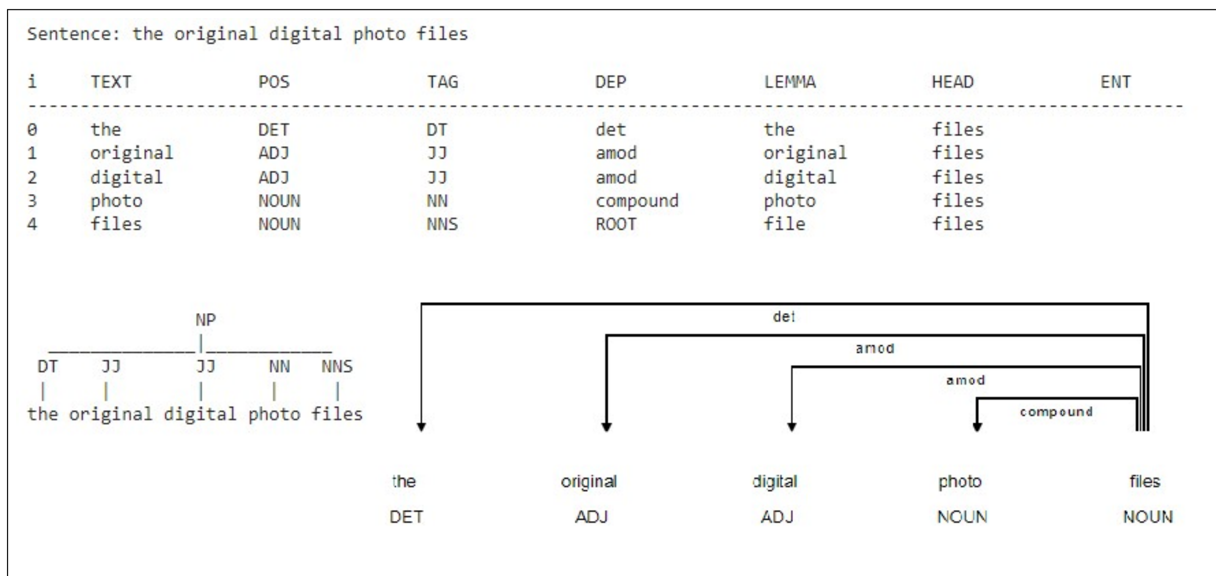
Listing E.2: Verb specification

```
1   verb_receive = Verb(
2       verb_str = 'receiving',
3       lemma = 'receive',
4       verb_type = VerbType.TRANSITIVE,
5       conjugations = VerbConjugations(
6           present_singular = 'receive',
7           present_plural = 'receives',
8           past = 'received',
9           continuous = 'receiving'
10      )
11  )
```

Finally, we need the direct object ('the original digital photo files'). As with the subject, an initial step is to check if this text corresponds with any assets or roles in the set of declarations. It does not, so we move on to more complex processing. We first run the text through SpaCy's NLP pipeline, which performs many of the steps discussed in Section 2.2, including tokenization, entity extraction, and POS-tagging. SpaCy is flexible in that it allows one to add new steps into the pipeline. We add a Python implementation of the Berkeley Neural Parser [3], which is able to perform consituency parsing, giving us a wider set of linguistic properties to work with. The result of this parsing is a set of tokens, dependency relationships and constituents, as shown in Fig E.1.

From this parse, it is fairly straightforward to extract the components required for our NounPhrase, shown in Listing E.3. We extract the head word ('files') as the head property. The is_plural property checks the tag of the head for plurality (NNS refers to a plural noun). We know it is not a role, since it was not found in the contract declarations. The determiner is identified as 'the', and the adjectives are identified by considering the tags labeled as adjectives (JJ) and also the dependencies labeled as 'compound' (e.g., 'photo'). Finally, to identify the asset type, we have a separate asset type extraction module, which looks for any *entities* found amongst the parsed tokens (e.g., locations, dates, organizations). If no entities are found, then the capitalized form of the head noun

[1] https://spacy.io/api/lemmatizer
[2] https://pypi.org/project/mlconjug3/
[3] https://pypi.org/project/benepar/



Sentence: the original digital photo files

i	TEXT	POS	TAG	DEP	LEMMA	HEAD	ENT
0	the	DET	DT	det	the	files	
1	original	ADJ	JJ	amod	original	files	
2	digital	ADJ	JJ	amod	digital	files	
3	photo	NOUN	NN	compound	photo	files	
4	files	NOUN	NNS	ROOT	file	files	

Figure E.1: SpaCy's parse of the direct object

is used ('Files'). This will result in the creation of a new asset on the domain update extraction step of the mapping.

Listing E.3: Direct object specification

```
photo_files = NounPhrase(
    str_val = 'the original digital photo files',
    head = 'files',
    is_plural = True,
    is_role = False,
    determiner = 'the',
    adjs = ['original', 'digital', 'photo'],
    asset_type = 'Files'
)
```

We now have the three components required for our full CustomEvent. These properties will be used for naming purposes and further processing in the norm update and domain update extraction steps. Events can also consist of components such as adverbs, predicates, and prepositional phrases. These components are processed in a similar way to what is described above.

There are many potential expansions of the event specification in SymboleoNLP, which would necessitate more complex processing. We emphasize that many of the approaches described here are based on machine learning, and are thus *probabilistic*. In certain cases, this is a necessity if we want to accomplish certain NLP tasks, and this highlights a use-case of leveraging ML-based techniques for a rules-based approach by using them for very specific purposes (e.g., conjugation, lemmatization). Future work may involve expanding this functionality to incorporate further models, as well as performing formal validation of these modules to assess their reliability in this context.

Appendix F

Contract Specification Parameter Processing

In Section 5.3, we discussed the mapping of an extracted pattern class to a set of norm updates and domain updates. We offered concrete examples of how a new event maps to a new domain event and a corresponding declaration. It is also possible for certain components of a pattern class to correspond to new contract specification parameters (CSPs) on the contract declaration itself. In traditional contract templates, many of these CSPs might align with FITB templates. This work focuses on contract templates that go *beyond* FITB templates, which we've noted would be trivial to process. We are concerned with processing more complex linguistic structures. Where a FITB template may have a simple date parameter (e.g., '`PartyA must perform event X before [DATE]`', our templates allow for more flexibility (e.g., '`PartyA must perform event X [P1]`'. The refinement for P1 can take many linguistic forms, as defined by our CNL.

We have defined our CNL to allow for dynamic values for certain components of a pattern class. For example, the DATE pattern variable in the P_BEFORE_S DATE pattern class is a dynamic input unit that can be filled in by the user. In this thesis, our examples have mainly illustrated cases where the user enters concrete date values into this dynamic input (e.g., '`PartyA must perform event X` *`before March 30, 2024`*'). The resulting refined contract involves adding the concrete date as an argument for a CSP in the contract instantiation (e.g., `contract = MyContract(target_date = 'March 30,2024')`, and then referencing that variable in the Symboleo obligation: `O(partyA, partyB, true, ShappensBefore(evt_x, target_date)`. In a sense, this operation results in refining the initial contract template into a FITB template. The resulting NL is '`PartyA must perform event X before [TARGET_DATE]`'.

To allow for this type of flexibility in our application, we configure certain dynamic units to have the option of the user entering an explicit CSP. If the user enters text that adheres to a specific syntax into certain dynamic units (such as the DATE or DOBJ units), then the internal processing will be slightly different. The syntactic pattern built into the tool is that the value must be enclosed in square brackets, underscores are used instead of spaces, and all letters are capitalized (e.g., [DUE_DATE], [TEST_OBJECT], [PURCHASE_AMOUNT]).

From a processing perspective, input that corresponds to a new CSP is much more straightforward. We do not require any NLP libraries such as SpaCy to look for parts-of-speech or dependency types. For example, suppose the user enters a refinement such as '`before PartyA sends [TEST_OBJECT]`'. This contains a custom event, which has a direct object (DOBJ) with a value of '[TEST_OBJECT]', which corresponds to our CSP syntax. By doing so, the user has turned a more flexible contract template into a FITB template. We have mentioned the complexity associated with mapping the custom event components into Symboleo constructs (such as declarations and domain objects), but this case is much simpler. From this refinement, we want to obtain the following results:

- **Add CSP:** `Contract(test_object: String, ...)`
- **New Domain Event:** `SendTestObject: sending_agent: Role, test_object: String;`
- **Declaration:** `evt_send_test_object: SendTestObject with sending_agent := PartyA, test_object := TEST_OBJECT`
- **Norm refinement:** `WhappensBefore(evt_x, evt_send_test_object)`

The type in this example is simply a string, but the tool can accommodate other types as well. If the text is entered on a DATE unit, then the type will be 'Date'. If keywords such as 'amount', 'price', or 'fee' are used, then the type will be 'Number'. There is also a case for defining a new *asset* from the text. For example, we may want to create an asset called `TestObject`. There may also be cases where a newly introduced asset should be framed as a new enumeration or a new type on an existing enumeration. More sophisticated processing such as this is left to future work, as is the more precise validation on the input of new CSPs and the inclusion of other dynamic unit types as candidates as potential CSP inputs.

Bibliography

[1] Ali Alzubaidi, Ellis Solaiman, Pankesh Patel, and Karan Mitra. Blockchain-based sla management in the context of iot. *IT Professional*, 21(4):33–40, 2019. doi:10.1109/MITP.2019.2909216.

[2] Elliott Ash, Jeff Jacobs, Bentley MacLeod, Suresh Naidu, and Dominik Stammbach. Unsupervised extraction of workplace rights and duties from collective bargaining agreements. In *2020 International Conference on Data Mining Workshops (ICDMW)*, pages 766–774, 2020. doi:10.1109/ICDMW51313.2020.00112.

[3] Tara Athan, Harold Boley, Guido Governatori, Monica Palmirani, Adrian Paschke, and Adam Wyner. OASIS LegalRuleML. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law*, ICAIL '13, page 3–12, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320801. doi:10.1145/2514601.2514603.

[4] Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The Berkeley FrameNet project. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*, pages 86–90, Montreal, Quebec, Canada, August 1998. Association for Computational Linguistics. doi:10.3115/980845.980860.

[5] Gavina Baralla, Andrea Pinna, Roberto Tonelli, Michele Marchesi, and Simona Ibba. Ensuring transparency and traceability of food local products: A blockchain application to a Smart Tourism Region. *Concurrency and Computation: Practice and Experience*, 33(1):e5857, 2021. doi:https://doi.org/10.1002/cpe.5857.

[6] Flávia Barros, Laís Neves, Erica Hori, and Dante Torres. The ucsCNL: A controlled natural language for use case specifications. In *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 250–253, 01 2011. URL `https://repositorio.ufpe.br/handle/123456789/2393`.

[7] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi:10.1109/72.279181.

[8] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34 – 43, 2001. doi:10.1038/scientificamerican0501-34.

[9] Carlo Biagioli, Enrico Francesconi, Andrea Passerini, Simonetta Montemagni, and Claudia Soria. Automatic semantics extraction in law documents. In *Proceedings of the 10th International Conference on Artificial Intelligence and Law*, ICAIL '05, page 133–140, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930817. doi:10.1145/1165485.1165506.

[10] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. O'Reilly Media, Sebastopol, CA, July 2009.

[11] Tolga Bolukbasi, Kai-Wei Chang, James Y Zou, Venkatesh Saligrama, and Adam T Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL `https://proceedings.neurips.cc/paper_files/paper/2016/file/a486cd07e4ac3d270571622f4f316ec5-Paper.pdf`.

[12] Stefano Borgo and Claudio Masolo. *Ontological Foundations of DOLCE*, pages 279–295. Springer Netherlands, Dordrecht, 2010. ISBN 978-90-481-8847-5. doi:10.1007/978-90-481-8847-5_13.

[13] Travis D. Breaux and Annie I. Antón. Analyzing goal semantics for rights, permissions, and obligations. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 177–186, 2005. doi:10.1109/RE.2005.12.

[14] Travis D. Breaux, Annie I. Antón, and Jon Doyle. Semantic parameterization: A process for modeling domain descriptions. *ACM Trans. Softw. Eng. Methodol.*, 18(2), nov 2008. ISSN 1049-331X. doi:10.1145/1416563.1416565.

[15] Igor Buzhinsky. Formalization of natural language requirements into temporal logics: a survey. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 400–406, 2019. doi:10.1109/INDIN41052.2019.8972130.

[16] Ilias Chalkidis, Ion Androutsopoulos, and Achilleas Michos. Obligation and prohibition extraction using hierarchical RNNs. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 254–259, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi:10.18653/v1/P18-2041.

[17] Ilias Chalkidis, Manos Fergadiotis, Prodromos Malakasiotis, Nikolaos Aletras, and Ion Androutsopoulos. LEGAL-BERT: The muppets straight out of law school. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2898–2904, Online, November 2020. Association for Computational Linguistics. doi:10.18653/v1/2020.findings-emnlp.261.

[18] Ilias Chalkidis, Abhik Jana, Dirk Hartung, Michael Bommarito, Ion Androutsopoulos, Daniel Katz, and Nikolaos Aletras. LexGLUE: A benchmark dataset for legal language understanding in English. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4310–4330, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi:10.18653/v1/2022.acl-long.297.

[19] Boqi Chen, Kua Chen, Yujing Yang, Shabnam Hassaniahari, Daniel Amyot, Lysanne Lessard, Gunter Mussbacher, Mehrdad Sabetzadeh, and Dániel Varró. On the use of GPT-4 for creating goal models: An exploratory study. In *13th International Model-Driven Requirements Engineering Workshop (MoDRE)*. IEEE CS, 2023.

[20] Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. Stipula: a domain specific language for legal contracts. In *Int. Workshop Programming Languages and the Law (ProLaLa)*, 2002. URL `https://www.math.unipd.it/~crafa/Pubblicazioni/ProLaLa22.pdf`.

[21] Cleyton Mário de Oliveira Rodrigues, Frederico Luiz Gonçalves de Freitas, Emanoel Francisco Spósito Barreiros, Ryan Ribeiro de Azevedo, and Adauto-Trigueiro de Almeida-Filho. Legal ontologies over time: A systematic mapping study. *Expert Systems with Applications*, 130:12–30, 2019. ISSN 0957-4174. doi:10.1016/j.eswa.2019.04.009.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi:10.18653/v1/N19-1423.

[23] Edsger W. Dijkstra. Notes on structured programming, 1970. URL `https://dl.acm.org/doi/pdf/10.5555/1243380`.

[24] Vimal Dwivedi, Alex Norta, Alexander Wulf, Benjamin Leiding, Sandeep Saxena, and Chibuzor Udokwu. A formal specification smart-contract language for legally binding decentralized autonomous organizations. *IEEE Access*, 9:76069–76082, 2021. doi:10.1109/ACCESS.2021.3081926.

[25] Mirna El Ghosh and Habib Abdulrab. Towards a pattern-based core model of events in the legal domain. In *CEUR Workshop Proceedings*, volume 2518, 2019. URL `https://ceur-ws.org/Vol-2518/paper-CREOL2.pdf`.

[26] Milva Finnegan. From a natural language to a controlled contract language. *Jusletter IT*, May 2018. URL `https://ssrn.com/abstract=3184366`.

[27] Markus Fockel and Jörg Holtmann. A requirements engineering methodology combining models and controlled natural language. In *2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE)*, pages 67–76, 2014. doi:10.1109/MoDRE.2014.6890827.

[28] Markus Fockel and Jörg Holtmann. ReqPat: Efficient documentation of high-quality requirements using controlled natural language. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pages 280–281, 2015. doi:10.1109/RE.2015.7320438.

[29] Enrico Francesconi. The “Norme in Rete” project: Standards and tools for Italian legislation. *International Journal of Legal Information*, 34(2):358–376, 2006. doi:10.1017/S0731126500001517.

[30] Norbert E. Fuchs. Understanding texts in Attempto Controlled English. *Frontiers in Artificial Intelligence and Applications*, 304:75–84, 2018. doi:10.3233/978-1-61499-904-1-75.

[31] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English — not just another logic specification language. In Pierre Flener, editor, *Logic-Based Program Synthesis and Transformation*, pages 1–20. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-48958-0. doi:10.1007/3-540-48958-4_1.

[32] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for knowledge representation. In *Reasoning Web: 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures*, pages 104–124. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-85658-0. doi:10.1007/978-3-540-85658-0_3.

[33] Ruka Funaki, Yusuke Nagata, Kohei Suenaga, and Shinsuke Mori. A contract corpus for recognizing rights and obligations. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pages 2045–2053, Marseille, France, May 2020. European Language Resources Association. ISBN 979-10-95546-34-4. URL `https://aclanthology.org/2020.lrec-1.251`.

[34] Leilei Gan, Kun Kuang, Yi Yang, and Fei Wu. Judgment prediction via injecting legal knowledge into neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14):12866–12874, May 2021. doi:10.1609/aaai.v35i14.17522.

[35] Aldo Gangemi, Maria-Teresa Sagri, and Daniela Tiscornia. *A Constructive Framework for Legal Ontologies*, pages 97–124. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-32253-5. doi:10.1007/978-3-540-32253-5_7.

[36] Shruti Gaur, Nguyen H. Vo, Kazuaki Kashihara, and Chitta Baral. Translating simple legal text to formal representations. In Tsuyoshi Murata, Koji Mineshima, and Daisuke Bekki, editors, *New Frontiers in Artificial Intelligence*, pages 259–273. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48119-6_19.

[37] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. Arsenal: Automatic requirements specification extraction from natural language. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods*, pages 41–46, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-40648-0_4.

[38] Hila Gonen and Yoav Goldberg. Lipstick on a pig: Debiasing methods cover up systematic gender biases in word embeddings but do not remove them. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 609–614, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi:10.18653/v1/N19-1061.

[39] Cristine Griffo. UFO-L: A core ontology of legal concepts built from a legal relations perspective. In *Doctoral Consortium - DC3K, (IC3K 2015)*, pages 13–20. SciTePress, 2015. doi:10.5220/0005647700130020.

[40] Robin Gröpler, Viju Sudhi, Emilio José Calleja García, and Andre Bergmann. NLP-based requirements formalization for automatic test case generation. In *CEUR Workshop Proceedings*, volume 2951, page 18 – 30, 2021. URL `https://ceur-ws.org/Vol-2951/paper15.pdf`.

[41] Giancarlo Guizzardi, Gerd Wagner, João Paulo Andrade Almeida, and Renata S.S. Guizzardi. Towards ontological foundations for conceptual modeling: The unified foundational ontology (UFO) story. *Applied Ontology*, 10:259–271, 2015. ISSN 1875-8533. doi:10.3233/AO-150157. 3-4.

[42] Rajaa El Hamdani, Majd Mustapha, David Restrepo Amariles, Aurore Troussel, Sébastien Meeùs, and Katsiaryna Krasnashchok. A combined rule-based and machine learning approach for automated GDPR compliance checking. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Law*, ICAIL '21, page 40–49, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385268. doi:10.1145/3462757.3466081.

[43] Dan Hendrycks, Collin Burns, Anya Chen, and Spencer Ball. CUAD: An expert-annotated NLP dataset for legal contract review, 2021. URL `https://doi.org/10.48550/arXiv.2103.0626`.

[44] Juliza Hidayati, Rini Vamelia, Jihaad Hammami, and Endri Endri. Transparent distribution system design of halal beef supply chain. *Uncertain Supply Chain Management*, 11(1):31–40, 2023. doi:10.5267/j.uscm.2022.12.003.

[45] Stefan Hoefler and Alexandra Bünzli. Designing a controlled natural language for the representation of legal norms. In *CNL 2010: Second Workshop on Controlled Natural Languages*, volume 622 of *CEUR Workshop Proceedings*, 2010. doi:10.5167/uzh-35842.

[46] Rinke Hoekstra, Joost Breuker, Marcello Di Bello, and Alexander Boer. The LKIF core ontology of basic legal concepts. In Pompeu Casanovas, Maria Angela Biasiotti, Enrico Francesconi, and Maria-Teresa Sagri, editors, *Proceedings of the 2nd Workshop on Legal Ontologies and Artificial Intelligence Techniques June 4th, 2007, Stanford University, Stanford, CA, USA*, volume 321 of *CEUR Workshop Proceedings*, pages 43–63. CEUR-WS.org, 2007. URL `https://ceur-ws.org/Vol-321/paper3.pdf`.

[47] Wesley Newcomb Hohfeld. Fundamental legal conceptions as applied in judicial reasoning. *The Yale Law Journal*, 26(8):710–770, 1917. ISSN 00440094. URL `http://www.jstor.org/stable/786270`.

[48] Jörg Holtmann, Jan Meyer, and Markus von Detten. Automatic validation and correction of formalized, textual requirements. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 486–495, 2011. doi:10.1109/ICSTW.2011.17.

[49] Rodney Huddleston, Geoffrey K. Pullum, and Brett Reynolds. *A Student's Introduction to English Grammar*. Cambridge University Press, 2 edition, 2021. doi:10.1017/9781009085748.

[50] Sarthak Jain and Byron C. Wallace. Attention is not Explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3543–3556, Stroudsburg, PA, USA, 2019. Association for Computational Linguistics. doi:10.18653/v1/N19-1357.

[51] Vivek Joshi, Preethu Rose Anish, and Smita Ghaisas. Domain adaptation for an automated classification of deontic modalities in software engineering contracts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1275–1280, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi:10.1145/3468264.3473921.

[52] Hans Kamp and Uwe Reyle. A calculus for first order discourse representation structures. *Journal of Logic, Language and Information*, 5(3):297–348, Oct 1996. ISSN 1572-9583. doi:10.1007/BF00159343.

[53] Nadzeya Kiyavitskaya, Nicola Zeni, Travis D. Breaux, Annie I. Antón, James R. Cordy, Luisa Mich, and John Mylopoulos. Automating the extraction of rights and obligations for regulatory compliance. In Qing Li, Stefano Spaccapietra, Eric Yu, and Antoni Olivé, editors, *Conceptual Modeling - ER 2008*, pages 154–168. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87877-3. doi:10.1007/978-3-540-87877-3_13.

[54] Nadzeya Kiyavitskaya, Nicola Zeni, James R. Cordy, Luisa Mich, and John Mylopoulos. Cerno: Light-weight tool support for semantic annotation of textual documents. *Data and Knowledge Engineering*, 68(12):1470–1492, 2009. ISSN 0169-023X. doi:10.1016/j.datak.2009.07.012. Including Special Section: 21st IEEE International Symposium on Computer-Based Medical Systems (IEEE CBMS 2008) – Seven selected and extended papers on Biomedical Data Mining.

[55] Tobias Kuhn. A Survey and Classification of Controlled Natural Languages. *Computational Linguistics*, 40(1):121–170, 03 2014. ISSN 0891-2017. doi:10.1162/COLI_a_00168.

[56] Jan Ladleif and Mathias Weske. A unifying model of legal smart contracts. In Alberto H. F. Laender, Barbara Pernici, Ee-Peng Lim, and José Palazzo M. de Oliveira, editors, *Conceptual Modeling*, pages 323–337, Cham, 2019. Springer International Publishing. ISBN 978-3-030-33223-5. doi:10.1007/978-3-030-33223-5_27.

[57] Tomer Libal and Alexander Steen. Towards an executable methodology for the formalization of legal texts. In Mehdi Dastani, Huimin Dong, and Leon van der Torre, editors, *Logic and Argumentation*, pages 151–165, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44638-3. doi:10.1007/978-3-030-44638-3_10.

[58] Linguistic Data Consortium. *ACE (Automatic Content Extraction) English Annotation Guidelines for Events*, 5.4.3 2005.07.01 edition, 2005. URL https://www.ldc.upenn.edu/sites/www.ldc.upenn.edu/files/english-events-guidelines-v5.4.3.pdf.

[59] Abhishek Mahindrakar and Karuna Pande Joshi. Automating GDPR compliance using policy integrated blockchain. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 86–93, 2020. doi:10.1109/BigDataSecurity-HPSC-IDS49724.2020.00026.

[60] Gary Marcus. Hoping for the best as ai evolves. *Communications of the ACM*, 66(4): 6–7, 2023. URL https://dl.acm.org/doi/pdf/10.1145/3583078.

[61] Robert C Martin. *The clean coder: a code of conduct for professional programmers*. Pearson Education, 2011.

[62] Regan Meloche, Daniel Amyot, and John Mylopoulos. Towards legal contract formalization with controlled natural language templates. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*, pages 317–322, 2023. doi:10.1109/RE57278.2023.00042.

[63] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL https://doi.org/10.48550/arXiv.1301.3781.

[64] María Navas-Loro and Cristiana Santos. Events in the legal domain: first impressions. In *TERECOM@JURIX*, 2018. URL https://ceur-ws.org/Vol-2309/05.pdf.

[65] María Navas-Loro, Ken Satoh, and Víctor Rodríguez-Doncel. ContractFrames: Bridging the gap between natural language and logics in contract law. In Kazuhiro Kojima, Maki Sakamoto, Koji Mineshima, and Ken Satoh, editors, *New Frontiers in Artificial Intelligence*, pages 101–114, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31605-1. doi:10.1007/978-3-030-31605-1_9.

[66] James O'Neill, Paul Buitelaar, Cecile Robin, and Leona O'Brien. Classifying sentential modality in legal language: A use case in financial regulations, acts and directives. In *Proceedings of the 16th Edition of the International Conference on Articial Intelligence and Law*, ICAIL '17, page 159–168, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348911. doi:10.1145/3086512.3086528.

[67] Gordon J. Pace and Michael Rosner. A controlled language for the specification of contracts. In Norbert E. Fuchs, editor, *Controlled Natural Language*, pages 226–245. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14418-9. doi:10.1007/978-3-642-14418-9_14.

[68] Alireza Parvizimosaed. *Symboleo: Specification and Verification of Legal Contracts*. PhD thesis, University of Ottawa, 2022. URL http://dx.doi.org/10.20381/ruor-28399.

[69] Alireza Parvizimosaed, Marco Roveri, Aidin Rasti, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Model-checking legal contracts with SymboleoPC. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, pages 278–288, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3550355.3552449.

[70] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, Aidin Rasti, Ali Roudak, and John Mylopoulos. Specification and analysis of legal contracts with Symboleo. *Software and Systems Modeling*, 21(6):2395–2427, Dec 2022. ISSN 1619-1374. doi:10.1007/s10270-022-01053-6.

[71] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL `https://proceedings.mlr.press/v28/pascanu13.html`.

[72] Paulo F. Pires, Flávia C. Delicato, Raphael Cóbe, Thais Batista, Joseph G. Davis, and Joo Hee Song. Integrating ontologies, model driven, and CNL in a multi-viewed approach for requirements engineering. *Requirements Engineering*, 16(2):133–160, Jun 2011. ISSN 1432-010X. doi:10.1007/s00766-011-0116-1.

[73] Aidin Rasti. From Symboleo to smart contracts - a code generator. Master's thesis, University of Ottawa, 2023. URL `http://dx.doi.org/10.20381/ruor-28394`.

[74] Aidin Rasti, Daniel Amyot, Alireza Parvizimosaed, Marco Roveri, Luigi Logrippo, Amal Ahmed Anda, and John Mylopoulos. Symboleo2SC: From legal contract specifications to smart contracts. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, pages 300–310, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3550355.3552407.

[75] Emanuel Regnath and Sebastian Steinhorst. SmaCoNat: Smart contracts in natural language. In *2018 Forum on Specification and Design Languages (FDL)*, pages 5–16, 2018. doi:10.1109/FDL.2018.8524068.

[76] Abhishek Sainani, Preethu Rose Anish, Vivek Joshi, and Smita Ghaisas. Extracting and classifying requirements from software engineering contracts. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 147–157, 2020. doi:10.1109/RE48521.2020.00026.

[77] Eder J. Scheid, Bruno B. Rodrigues, Lisandro Z. Granville, and Burkhard Stiller. Enabling dynamic SLA compensation using blockchain-based smart contracts. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 53–61, 2019. URL `https://ieeexplore.ieee.org/abstract/document/8717859`.

[78] Murray Shanahan. *The Event Calculus Explained*, pages 409–430. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-48317-5. doi:10.1007/3-540-48317-9_17.

[79] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. Symboleo: Towards a specification language for legal contracts. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 364–369, 2020. doi:10.1109/RE48521.2020.00049.

[80] Seyed Sepehr Sharifi. Smart contracts: From formal specification to blockchain code. Master's thesis, University of Ottawa, 2020. URL `http://dx.doi.org/10.20381/ruor-25092`.

[81] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. The curse of recursion: Training on generated data makes models forget, 2023. URL `https://arxiv.org/abs/2305.17493`.

[82] Akhilesh Kumar Singh and Zahid Raza. A framework for IoT and blockchain based smart food chain management system. *Concurrency and Computation: Practice and Experience*, 35(4):e7526, 2023. doi:https://doi.org/10.1002/cpe.7526.

[83] Amin Sleimi, Nicolas Sannier, Mehrdad Sabetzadeh, Lionel Briand, and John Dann. Automated extraction of semantic legal metadata using natural language processing. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 124–135, 2018. doi:10.1109/RE.2018.00022.

[84] Michele Soavi, Nicola Zeni, John Mylopoulos, and Luisa Mich. ContracT – from legal contracts to formal specifications: Preliminary results. In Jānis Grabis and Dominik Bork, editors, *The Practice of Enterprise Modeling*, pages 124–137, Cham, 2020. Springer International Publishing. ISBN 978-3-030-63479-7. doi:10.1007/978-3-030-63479-7_9.

[85] Ghanem Soltana, Nicolas Sannier, Mehrdad Sabetzadeh, and Lionel C Briand. Model-based simulation of legal policies: Framework, tool support, and validation. *Software & Systems Modeling*, 17:851–883, 2018. doi:10.1007/s10270-016-0542-0.

[86] Evangelia Spiliopoulou, Eduard Hovy, and Teruko Mitamura. Event detection using frame-semantic parser. In *Proceedings of the Events and Stories in the News Workshop*, pages 15–20, Vancouver, Canada, August 2017. Association for Computational Linguistics. doi:10.18653/v1/W17-2703.

[87] Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997. URL `https://doi.org/10.5210/fm.v2i9.548`.

[88] Takaaki Tateishi, Sachiko Yoshihama, Naoto Sato, and Shin Saito. Automatic smart contract generation using controlled natural language and template. *IBM Journal of Research and Development*, 63(2/3):6:1–6:12, 2019. doi:10.1147/JRD.2019.2900643.

[89] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. EFLINT: A domain-specific language for executable norm specifications. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, page 124–136, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381741. doi:10.1145/3425898.3426958.

[90] L Thomas van Binsbergen, Milen G Kebede, Joshua Baugh, Tom Van Engers, and Dannis G van Vuurden. Dynamic generation of access control policies from social policies. *Procedia Computer Science*, 198:140–147, 2022. doi:10.1016/j.procs.2021.12.221.

[91] Tom Van Engers, Alexander Boer, Joost Breuker, André Valente, and Radboud Winkels. *Ontologies in the Legal Domain*, pages 233–261. Springer US, Boston, MA, 2008. ISBN 978-0-387-71611-4. doi:10.1007/978-0-387-71611-4_13.

[92] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[93] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models, 2023. URL `https://arxiv.org/abs/2305.19234`.

[94] Adam Wyner and Wim Peters. On rule extraction from regulations. *Frontiers in Artificial Intelligence and Applications*, 235:113–122, 2011. doi:10.3233/978-1-60750-981-3-113.

[95] Adam Wyner, Krasimir Angelov, Guntis Barzdins, Danica Damljanovic, Brian Davis, Norbert Fuchs, Stefan Hoefler, Ken Jones, Kaarel Kaljurand, Tobias Kuhn, Martin Luts, Jonathan Pool, Mike Rosner, Rolf Schwitter, and John Sowa. On controlled natural languages: Properties and prospects. In Norbert E. Fuchs, editor, *Controlled Natural Language*, pages 281–289. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14418-9. doi:10.1007/978-3-642-14418-9_17.

[96] Aya Zaki-Ismail, Mohamed Osama, Mohamed Abdelrazek, John Grundy, and Amani Ibrahim. RCM-extractor: an automated NLP-based approach for extracting a semi formal representation model from natural language requirements. *Automated Software Engineering*, 29(1):10, Dec 2021. ISSN 1573-7535. doi:10.1007/s10515-021-00312-y.

[97] Nicola Zeni, Nadzeya Kiyavitskaya, Luisa Mich, James R. Cordy, and John Mylopoulos. GaiusT: supporting the extraction of rights and obligations for regulatory compliance. *Requirements Engineering*, 20(1):1–22, Mar 2015. ISSN 1432-010X. doi:10.1007/s00766-013-0181-8.

[98] Jieyu Zhao, Tianlu Wang, Mark Yatskar, Ryan Cotterell, Vicente Ordonez, and Kai-Wei Chang. Gender bias in contextualized word embeddings. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 629–634, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi:10.18653/v1/N19-1064.