INTERPRETABLE MODELS FOR INFORMATION EXTRACTION

by

Marco Antonio Valenzuela Escárcega

______________________________
Creative Commons Attribution-No Derivative Works 3.0 License

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2016
As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Marco Antonio Valenzuela Escárcega, titled Interpretable Models for Information Extraction and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Mihai Surdeanu
Date: 18 April 2016

Clayton Morrison
Date: 18 April 2016

Alon Efrat
Date: 18 April 2016

Emek Demir
Date: 18 April 2016

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Mihai Surdeanu
Date: 18 April 2016
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of the source is made. This work is licensed under the Creative Commons Attribution-No Derivative Works 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

SIGNED: Marco Antonio Valenzuela Escárcega
I have been very fortunate to have met many brilliant people during my PhD. They have guided me and supported me during both the fun and the difficult times. I have learned a lot from all of them.

First and foremost, I would like to thank my advisor, Professor Mihai Surdeanu. In the years I have worked for him, he has taught me to be a better researcher, to set and achieve my goals, to better communicate my ideas, and overall to be a better person. I feel very fortunate to have had the opportunity to work with and learn from him, and I hope that I can continue to do it in the future. I will always be in debt with him.

I would also like to thank my dissertation committee, Professors Clayton Morrison, Alon Efrat, and Emek Demir. I am grateful for all their guidance, both about the dissertation and about my future goals.

Thanks to Dr. Paul Cohen for believing in me when I had nothing to show for myself and accepting me as one of his students. I learned a lot during my time working for him.

Thanks to Dr. Dino Pardo for encouraging me to pursue a PhD. He believed in me and guided me during the beginning of the program. He has always supported me and has been a mentor and a friend.

Thanks to my friends at the computational language understanding lab. Especially to my linguist friends: Gustave Hahn-Powell, Rebecca Sharp, and Dane Bell. They have become like a second family who has always supported me and given me their advice and their friendship. I will always treasure the moments we have shared together. Also thanks to Enrique Noriega Atala, Peter Jansen, and Tom Hicks. Thanks to all of them working in the lab has been so much fun.

The members of the old lab: Wesley Kerr, Daniel Hewlett, Anh Tran, Derek Green, Antons Rebguns, and Jeremy Wright. They welcomed me when I arrived to the program. I learned a lot from all of them.

Lastly, I would like to thank my family. Thanks to my parents for all the sacrifices they have made during the years for my education and my future. And thanks to my sister for always encouraging me and believing in me.
DEDICATION

A mis padres.
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... 9

LIST OF TABLES .......................................................... 11

ABSTRACT ................................................................. 12

CHAPTER 1 INTRODUCTION ............................................... 14  
  1.1 Motivation ......................................................... 14
  1.2 The Information Extraction Task ............................ 16
  1.3 Interpretability .................................................. 17
  1.4 Contributions ................................................... 18
  1.5 Overview ......................................................... 19

CHAPTER 2 DESIGN OF THE INFORMATION EXTRACTION FRAMEWORK ........................................ 20  
  2.1 Motivation ......................................................... 20
  2.2 A Walkthrough Example ....................................... 22
  2.3 Rule Language ................................................... 26  
     2.3.1 A Gentle Introduction to YAML .......................... 27
     2.3.2 Rules ....................................................... 29
     2.3.3 Token Patterns .......................................... 31
     2.3.4 Dependency Patterns ................................. 39
     2.3.5 Building a Grammar .......................... 48
  2.4 Mentions, or the Output of Rules ........................... 53  
     2.4.1 TextBoundMention .................................. 54
     2.4.2 RelationMention .................................. 54
     2.4.3 EventMention ......................................... 54

CHAPTER 3 IMPLEMENTATION OF THE INFORMATION EXTRACTION FRAMEWORK .......................... 55  
  3.1 Overview ........................................................ 55
  3.2 Extractor ........................................................ 55  
     3.2.1 Token Extractor ....................................... 57
     3.2.2 Dependency Extractor .............................. 57
     3.2.3 Actions ................................................ 57
  3.3 Extractor Engine ................................................ 58
# Table of Contents – Continued

3.3.1 State ...................................................... 59
3.4 Patterns ..................................................... 60
  3.4.1 Token Pattern ......................................... 60
  3.4.2 Dependency Pattern .................................. 66

Chapter 4 Application to Bioinformatics ......................... 73
  4.1 Motivation ............................................... 73
  4.2 REACH ................................................... 75
    4.2.1 Preprocessing ....................................... 76
    4.2.2 Entity Extraction ................................... 76
    4.2.3 Simple Events ....................................... 78
    4.2.4 Nested Events ....................................... 80
    4.2.5 Complex Natural Language Phenomena ............... 82
  4.3 Evaluation ............................................... 84

Chapter 5 Learning Interpretable Models ....................... 94
  5.1 The BioNLP’09 Shared Task ............................... 94
  5.2 The statistical model ................................... 95
    5.2.1 Trigger extraction .................................. 96
    5.2.2 Relation extraction .................................. 97
    5.2.3 Feature to rule translation limitations ............. 98
  5.3 Converting the statistical model to rules ............... 98
    5.3.1 Converting features to rules ......................... 98
    5.3.2 Converting weights to votes ......................... 99
  5.4 Model editing ............................................ 100
  5.5 Evaluation ............................................... 101

Chapter 6 Related Work ....................................... 107
  6.1 Information extraction with machine learning .......... 107
    6.1.1 Supervised .......................................... 107
    6.1.2 Semi Supervised .................................... 109
    6.1.3 Unsupervised ....................................... 112
  6.2 Rule Based IE ........................................... 112
  6.3 Conclusion ............................................... 116

Chapter 7 Conclusions ......................................... 117
  7.1 Summary .................................................. 117
  7.2 Future Work .............................................. 117
    7.2.1 Assembly of knowledge graph ......................... 118
    7.2.2 Rule learning ........................................ 118
# TABLE OF CONTENTS – Continued

7.2.3 ODIN Extensions ................................. 120

APPENDIX A Grammars .................................. 122

A.1 Token Constraint Grammar ............................. 122
A.2 Token Pattern Grammar ................................. 123
A.3 Dependency Pattern Grammar ............................ 125

APPENDIX B ODIN API ................................... 127

APPENDIX C Customizing Rule Output with Actions .......................... 133

REFERENCES .............................................. 135
| 2.1 | A sentence containing three proteins and two kinds of interactions among them: ubiquitinations and negative regulations. | 22 |
| 2.2 | Example of a sentence that has been annotated by an NLP stack. | 24 |
| 2.3 | ODIN output mentions. | 26 |
| 2.4 | YAML indicator characters | 29 |
| 2.5 | Comparison of greedy (default behavior) and lazy (?) quantifiers. | 35 |
| 2.6 | These sentences show some of the infinite syntactic variation describing a dance between two entities. | 39 |
| 2.7 | The relational-dependency parse for the sentences in Figure 2.6. | 40 |
| 2.8 | The structured output of the rule in Listing 2.10. | 41 |
| 2.9 | A simple sentence with its corresponding dependency parse tree. | 41 |
| 2.10 | The sentence used by the rules shown in Listings 2.13, 2.14, & 2.15. We are assuming that a different rule created a Location mention for every location NE. | 44 |
| 3.1 | Construction of a composite alternation NFA. | 61 |
| 3.2 | Composite NFA for the Kleene star. | 62 |
| 3.3 | Composite NFA representing optionality of a pattern. | 63 |
| 3.4 | Composite NFA representing a pattern to be matched one or more times (Kleene plus). | 63 |
| 3.5 | A sequence of VM threads where each thread represents a state of the NFA. Thread 4 represents a thread bundle, which is a nesting of several thread sequences. | 66 |
| 3.6 | Example of an input sentence to the grammar shown in Listing 3.11. | 69 |
| 4.1 | PubMed indexed publications per year. Note that these publication counts are not cumulative. | 74 |
| 4.2 | Sentence with nested controls | 84 |
| 4.3 | Control with reversed polarity. | 84 |
| 4.4 | Summary of the Big Mechanism evaluation. Team 4 was an ensemble system consisting of REACH and Team 4(B). | 89 |
| 4.5 | Number of rule templates in REACH’s grammar | 91 |
| 4.6 | Distribution of the number of entity and event matches per instantiated rule. | 92 |
| 5.1 | BioNLP’09 event types. Secondary arguments are omitted since they are not involved in the “core event extraction” task. | 95 |
LIST OF FIGURES – Continued

5.2 Architecture of the statistical model for the BioNLP 2009 core event extraction task. ................................................................. 96
5.3 Example of rule built from the conjunction of the trigger “phosphorylation”, the outgoing nsubjpass, and the semantic argument constraint of Protein. ................................................................. 99
5.4 Discretization of the trigger weights learned by the logistic regression classifier. ................................................................. 100
5.5 Discretization of the relation weights learned by the logistic regression classifier. ................................................................. 101
5.6 Performance of the IE pipeline comprised of logistic regression with $L_2$ regularization across all event types. .............................. 102
5.7 Performance of the IE pipeline comprised of logistic regression with $L_1$ regularization across all event types. .............................. 102
5.8 Performance of the IE pipeline comprised of automatically extracted rules from the logistic regression classifiers across all event types. . 103
5.9 Performance of the IE pipeline comprised of automatically extracted rules from the logistic regression classifiers, after human intervention. 104
5.10 Learning curve showing the change in F1 performance as a function of the amount of training data. We compare the performance of both the logistic regression with $L_1$ regularization (shown in red) and the final rule-learning system prior to the expert modifications (shown in blue). ................................................................. 105

6.1 Example JAPE rule. From https://gate.ac.uk/sale/talks/gate-course-may10/track-1/module-3-jape/module-3-jape.pdf 114
6.2 Example XTDL rule. From http://sprout.dfki.de/XTDLSampleRule.html ................................................................. 114
# LIST OF TABLES

2.1 Overview of the fields of an ODIN rule. ........................................ 30  
2.2 An overview of the attributes that may be specified in a token constraint. .......................................................................................... 32  
2.3 Example of parentheses usage to change operator precedence. ........ 35  
2.4 An overview of the quantifiers supported by ODIN’s token patterns. 36  
2.5 An overview of the zero-width assertions supported by ODIN. These patterns do not consume tokens, but are useful to match patterns preceding/following the expression of interest. ..................... 37  
2.6 The four mentions produced by the dependency pattern shown in Listing 2.13. ................................................................. 45  
2.7 The single mention produced by the rule shown in Listing 2.14. ...... 45  
2.8 The six mentions produced by the rule shown in Listing 2.15. ....... 45  
2.9 An overview of the quantifiers supported by ODIN’s dependency patterns. .............................................................. 46  
2.10 An overview of the most important fields of the Mention class. ... 53  
3.2 Explanation of range quantifier compiler expansions for token patterns. 64  
3.4 Explanation of range quantifier compiler expansions for dependency patterns. ............................................................. 67  
3.1 Virtual machine instructions. .......................................................... 71  
3.3 Dependency pattern instructions ...................................................... 72  
4.1 Publicly available databases used to ground entities in the REACH system. ................................................................. 78  
4.2 Common syntactic variations shared among events. ...................... 79  
4.3 Results of the summer 2015 Big Mechanism evaluation. ............... 88  
4.4 Strict precision for specific event types as determined during the Big Mechanism 2015 evaluation. ................................. 90  
5.1 Representative examples of the rule changes suggested by linguistic experts. ................................................................. 106
ABSTRACT

There is an abundance of information being generated constantly, most of it encoded as unstructured text. The information expressed this way, although publicly available, is not directly usable by computer systems because it is not organized according to a data model that could inform us how different data nuggets relate to each other. Information extraction provides a way of scanning unstructured text and extracting structured knowledge suitable for querying and manipulation. Most information extraction research focuses on machine learning approaches that can be considered black boxes when deployed in information extraction systems.

We propose a declarative language designed for the information extraction task. It allows the use of syntactic patterns alongside token-based surface patterns that incorporate shallow linguistic features. It captures complex constructs such as nested structures, and complex regular expressions over syntactic patterns for event arguments. We implement a novel information extraction runtime system designed for the compilation and execution of the proposed language. The runtime system has novel features for better declarative support, while preserving practicality. It supports features required for handling natural language, like the preservation of ambiguity and the efficient use of contextual information. It has a modular architecture that allows it to be extended with new functionality, which, together with the language design, provides a powerful framework for the development and research of new ideas for declarative information extraction.

We use our language and runtime system to build a biomedical information extraction system. This system is capable of recognizing biological entities (e.g., genes, proteins, protein families, simple chemicals), events over entities (e.g., biochemical reactions), and nested events that take other events as arguments (e.g., catalysis). Additionally, it captures complex natural language phenomena like coreference and
hedging.

Finally, we propose a rule learning procedure to extract rules from statistical systems trained for information extraction. Rule learning combines the advantages of machine learning with the interpretability of our models. This enables us to train information extraction systems using annotated data that can then be extended and modified by human experts, and in this way accelerate the deployment of new systems that can still be extended or modified by human experts.
CHAPTER 1

INTRODUCTION

1.1 Motivation

Since the advent of the digital computer, and particularly since the inception of the Internet, there has been an unprecedented proliferation of digital information, most of which is encoded as unstructured data. It is estimated that up to 80% of the web consists of unstructured text\(^1\). This unstructured text takes many forms, from websites, emails, and tweets to scientific journals. The information expressed this way, although publicly available, is not directly usable by computer systems because it is not organized according to a data model that could inform us how different data nuggets relate to each other.

This problem is particularly prevalent in the scientific literature. Scientific fields get fragmented into subfields in order to deal with the high volume of new knowledge being generated (Casadevall and Fang, 2014). This scientific specialization has the unintended consequence of precluding the interaction between different scientific fields. Swanson (1986) coined the term “undiscovered public knowledge” to refer to knowledge that, although published and publicly available, remains undiscovered because the individual fragments have never been brought together.

Because of the high volume of scientific publications, both within and across fields, scientists can’t keep track of all the research being conducted that may be related to their area of work. We can envision a computer system, however, that could keep track of all the literature being published while searching for logical relations across the different knowledge fragments. This hypothetical system could suggest new scientific hypotheses by connecting seemingly unrelated facts,

and find information in published experimental reports that may be relevant to the formulated hypothesis, even if the experiments were conducted for a different reason (Blagosklonny and Pardee, 2002; Evans and Rzhetsky, 2010). Previous efforts have shown that new knowledge can be synthesized by finding connections between knowledge fragments (Swanson and Smalheiser, 1996, 1997).

In order to make connections between knowledge fragments, a computer system like the one envisioned should first extract the relevant fragments from the literature. This task is known as information extraction (IE). Current research mostly focuses on developing machine learning (ML) approaches to the IE task. ML systems have good performance, but they are usually black boxes that are hard to understand and modify (Sculley et al., 2014). This is not ideal for industry applications, where systems have to be maintained for long periods of time and the ability to make isolated improvements to functioning systems is a requirement. This is why rule-based solutions are more prevalent in the industry (Chiticariu et al., 2013).

We refer to these kind of systems as “interpretable models”, which must have two properties: a) humans should be able to understand the models, and b) humans should be able to modify them, either by correcting or extending them. For example, neural networks have neither of these two characteristics. Decision trees have the first property, since their functioning can be understood by the users, but they are not easy to modify.

In this dissertation we focus on the practical extraction of structured information from unstructured text by using interpretable models that can be adjusted to suit the needs of domain experts in different fields. For this purpose, we propose a framework that consists of a declarative language and a runtime system to build IE systems. Before we describe our contributions we describe information extraction and interpretability.
1.2 The Information Extraction Task

As we mentioned previously, the goal of IE is to scan unstructured data and extract the knowledge contained in it. The extracted knowledge is generally structured with respect to some data model that describes the different types of information fragments we are interested in, as well as the valid relations between them. The advantage of structured data is that it is suitable for automated querying and manipulation by a computer system. One notable exception to this requirement is open information extraction, which is concerned with the extraction of unspecified relations from large corpora (Banko et al., 2008). Open IE produces semi-structured data that does not conform to any particular data model, but instead can be said to be self-describing.

Most IE tasks can be split into two phases: detecting the entities mentioned in the text and finding associations between them. IE is usually domain-specific. In these situations we have an ontology that formally models the entities of interest and the relations between them (Guarino et al., 2009).

The main subtasks of IE are:

**Named Entity Recognition** (NER) The task of identifying relevant entities mentioned in text. The types of these entities varies depending on the domain of interest. In the open domain, these entities correspond to people, places, and organizations. They can also correspond to dates, currency, and other kinds of numeric expressions. In the biomedical domain, they are usually proteins, chemicals, and subcellular locations, among others.

**Coreference Resolution** The detection of anaphoric links between the extracted entities. An example of an anaphoric link found in a biomedical paper is “...we incubated GSK3β with excess Axin GBD protein to saturate its binding to GSK3β ...”. In this sentence, the preposition “its” refers to the “GSK3β” enzyme.

**Relation and Event Extraction** The process of identifying how the extracted
entities interact with each other. Relations describe how two entities relate to each other. Examples of relations in the open domain are “PERSON is-employed-by ORGANIZATION” or “PERSON was-born-in LOCATION”. Events are a generalization of relations that can involve one or more entities. They can also be nested, i.e., they involve other events. An example from the biomedical domain is “CYLD inhibits the ubiquitination of TRAF2”. This sentence contains two events: a) the ubiquitination of TRAF2, and b) the inhibition of the ubiquitination event by CYLD.

Machine learning approaches to information extraction rely on the availability of manually-annotated corpora. These corpora are not always available for the domains of interest. In these situations, extractors can be trained directly from existing databases, which are common in many domains. The process of generating weak labels from an existing database is known as distant supervision (Mintz et al., 2009).

1.3 Interpretability

Rule-based IE has long enjoyed wide adoption throughout industry, though it has remained largely ignored in academia, in favor of ML methods (Chiticariu et al., 2013). However, rule-based systems have several advantages over pure ML systems, including: (a) the rules are interpretable and thus suitable for rapid development and domain transfer; and (b) humans and machines can contribute to the same model.

These advantages mean that rule-based IE models can be read, understood, and modified by the user. This ability enables the user to both introduce domain knowledge to the model, and also make incremental and isolated enhancements to the model. ML models, on the other hand, are usually black boxes that incorporate many heterogeneous data sources into a single output. This data entanglement makes isolated improvements impossible. Sculley et al. (2014) refer to this situation as the CACE principle (i.e., changing anything changes everything).
Why then have rule-based systems failed to hold the attention of the academic community? One argument raised by Chiticariu et al. (2013) is that, despite notable efforts (Appelt and Onyshkevych, 1998; Levy and Andrew, 2006; Hunter et al., 2008; Cunningham et al., 2011; Chang and Manning, 2014), there is not a standard language for this task, or a “standard way to express rules”, which raises the entry cost for new rule-based systems.

1.4 Contributions

We propose one way of addressing this deficit with the following contributions:

1. A declarative language designed for the IE task. It allows the use of syntactic patterns alongside token-based surface patterns that incorporate shallow linguistic features. It captures complex constructs such as nested structures, and complex regular expressions over syntactic patterns for event arguments. The language is designed to be modular, i.e., new types of rules can be easily added to the language. We currently support rules based on syntactic and surface structures, and we plan extensions over abstract meaning representation (AMR) (Banarescu et al., 2012) and semantic roles (Surdeanu et al., 2008). Importantly, all of these types of rules can operate within the same grammar.

2. A novel IE runtime system designed for the compilation and execution of the above interpretable models. The runtime system has novel features for better declarative support, while preserving practicality. It supports features required for handling natural language, like the preservation of ambiguity and the efficient use of contextual information. It has a modular architecture that allows it to be extended with new functionality, which, together with the language design, provides a powerful framework for the development and research of new ideas for declarative IE.

3. A biomedical IE system built using our language and runtime system. This system is capable of recognizing biological entities (e.g., genes, proteins, pro-
tein families, simple chemicals), events over entities (e.g., biochemical reactions), and nested events that take other events as arguments (e.g., catalysis). Additionally, it captures complex natural language phenomena like coreference and hedging. This IE system had the best overall performance in a recent evaluation organized as part of DARPA’s Big Mechanism program (Cohen, 2015).

4. A rule learning procedure to extract rules from statistical systems trained for IE. Rule learning combines the advantages of machine learning with the interpretability of our models. This enables us to train IE systems using annotated data that can then be extended and modified by human experts, and in this way accelerate the deployment of new systems that can still be extended or modified by human experts. The ability to extract rules from a statistical system makes possible the use of learning schemes like distant supervision, which would facilitate deployment even further.

1.5 Overview

This dissertation is organized as follows: Chapter 2 addresses our first contribution by describing the design of the IE declarative language. Chapter 3 describes the IE runtime system implementation, which is our second contribution. Chapter 4 describes the construction of REACH, a real-world IE system tailored for the biomedical domain, which corresponds to our third contribution. Chapter 5 explores the automatic generation of ODIN rules by using established statistical methods, which corresponds to our fourth contribution. Chapter 6 describes previous work on both statistical and rule-based IE. Finally, Chapter 7 summarizes our results and proposes future research directions.
CHAPTER 2

DESIGN OF THE INFORMATION EXTRACTION FRAMEWORK

In this chapter we discuss the design of the ODIN information extraction framework.

The ODIN information extraction framework was developed by Valenzuela-Escárcega et al. (2015a,b, 2016).

2.1 Motivation

ODIN (Open Domain INformer) aims to address the issues raised in Section 1.3 with a novel information extraction (IE) language and framework. The design of ODIN followed the simplicity principles promoted by other natural language processing toolkits, such as Stanford’s CoreNLP, which aim to “avoid over-design”, “do one thing well”, and have a user “up and running in ten minutes or less” (Manning et al., 2014). For example, consider a domain that tracks people’s movement, as reported in the news. One may want to quickly write a domain grammar that captures events with the following arguments: (a) the subject of the verb “move” (and its synonyms) only if it has been identified as a PERSON by a named entity recognizer (NER), (b) the indirect object of the same verb that is dominated by the preposition “from” as the origin location, and (c) an indirect object dominated by the preposition “to” as the destination. ODIN captures such event patterns (and more) using a single declarative rule.

In particular, ODIN is:

Simple: Taking advantage of a syntactic dependency (SD) representation (de Marneffe and Manning, 2008), our IE language has a simple, declarative syntax for the extraction of n-ary events, which captures single or multi-word event predicates with lexical and morphological constraints, and event arguments with (generally) simple syntactic patterns and semantic constraints.
**Powerful:** Despite its simplicity, our IE framework can capture complex constructs when necessary, such as: (a) nested events\(^1\), and (b) complex regular expressions over syntactic patterns for event arguments. Inspired by Stanford’s Semgrex\(^2\), we have extended a standard regular expression language to describe patterns over directed graphs\(^3\), e.g., we introduce new `<` and `>` operators to specify the direction of edge traversal in the dependency graph. Finally, we allow for (c) optional arguments and multiple arguments with the same name.

**Robust:** To recover from unavoidable syntactic errors, SD patterns (such as the ones shown in the next section) can be freely mixed with token-based surface patterns, using a language inspired by the Allen Institute of Artificial Intelligence’s Tagger\(^4\). These patterns match against information extracted in our text processing pipeline\(^5\), namely a token’s part of speech, lemmatized form, named entity label, and the immediate incoming and outgoing edges in the SD graph.

**Fast:** Our IE runtime is fast because the ODIN runtime uses event trigger phrases (e.g., “move” for a moving event), which are captured with lexico-morphological patterns, as shallow filters to reduce the search space for pattern matching. That is, only when event triggers are detected is the matching of more complex syntactic patterns for arguments attempted. This guarantees quick executions. For example, in a real-world biochemical domain, ODIN processes an average of 110 sentences/second\(^6\) with a grammar of 211 rules on a laptop with an i7 CPU and 16GB of RAM.

This chapter is organized as follows. Section 2.2 introduces the ODIN rule language with a simple walkthrough example. Section 2.3 describes the complete rule

---

\(^1\)Events that take other events as arguments. See the walkthrough example in the next section.

\(^2\)nlp.stanford.edu/software/tregex.shtml

\(^3\)We currently use Stanford syntactic dependencies, but any other graph derived from text could be used. For example, one could use a graph that models semantic roles or abstract meaning representation.

\(^4\)https://github.com/allenai/taggers

\(^5\)https://github.com/clulab/processors

\(^6\)After the initial text processing pipeline that includes syntactic parsing.
language. Lastly, Section 2.4 describes ODIN mentions, which are Scala\textsuperscript{7} objects that store the output of rules.

2.2 A Walkthrough Example

Let's use the sentence in Figure 2.1 as a simple walkthrough example for an ODIN grammar in the biomedical domain. This particular sentence contains three proteins (i.e., CYLD, TRAF2, and TRAF6). We would like to build a grammar that finds molecular interactions of two kinds: protein ubiquitination (a kind of post-translational modification) and negative regulations of the ubiquitinations.

**CYLD** inhibits the ubiquitination of both **TRAF2** and **TRAF6**

Figure 2.1: A sentence containing three proteins and two kinds of interactions among them: ubiquitinations and negative regulations.

The minimal requirement for ODIN input is text that is tokenized into distinct words. However, as shown in Listing 2.1, ODIN rules frequently use syntax to capture patterns. It is therefore common that any text that will be scanned by ODIN be first annotated using standard natural language processing (NLP) tools such as text segmentation, part-of-speech (POS) tagging, named-entities recognition (NER), and syntactic parsing. We provide a brief description of each of these tasks below.

**Text Segmentation**

Text segmentation refers to the task of dividing text into meaningful units such as sentences and words. These tasks are commonly referred to as sentence splitting and tokenization, respectively.

**Part-of-speech tagging**

Part-of-speech (POS) tags are categories that capture syntactic functions of

\textsuperscript{7}ODIN is implemented in the Scala language. However, because Scala runs on the Java Virtual Machine (JVM), it plays well with other JVM languages, most notably Java.
words, e.g., nouns, verbs, and prepositions. POS tags are generally assigned using sequence models such as conditional random fields (CRF), described in Lafferty et al. (2001).

**Named-entity recognition**

Named-entity recognition (NER) is another sequence modeling task that identifies sections of text referring to mentions of entities of interest. Examples of such entities are names of people, places, and organizations (in an open-domain scenario), or proteins, simple chemicals and cellular locations (in the biomedical domain). Because named entities can be composed of multiple words, they are typically labeled using the BIO notation, first proposed by Ramshaw and Marcus (1999). This representation uses the prefixes $B -$ and $I -$ to denote words that are at the beginning or inside a group, respectively, and $O$ to denote tokens that don’t form part of any group.

**Syntactic parsing**

Lastly, syntactic parsing captures intra-sentence grammatical relations (e.g., subject, object). There are multiple possible representations for these relations. ODIN uses the Stanford typed dependencies representation (de Marneffe and Manning, 2008). Stanford dependencies consist of directed and labeled binary relations between two words: head and dependent. For example, the relation between a verb and its subject is encoded as the $nsubj$ relation where the verb is the head and the subject is the dependent. We recommend the use of the collapsed dependencies representation, where dependencies involving prepositions and conjunctions are collapsed in order to get direct connections between content words. This facilitates the development of extraction rules. Note, however, that ODIN is agnostic to which representation is used.

Figure 2.2 shows the sentence from Figure 2.1 after it has been annotated by an NLP stack. The text has been tokenized and the colored boxes above each word

---

display its POS tag. The transparent boxes below each word show named-entity tags in BIO notation. The arrows connecting the tokens indicate the directed edges of the dependency graph, connecting the head to the dependent. Note that any NLP tools could be used to implement these preprocessing steps. This will be important in Chapter 4, where we use ODIN to build a biomedical IE system.

![Dependency Graph Example](image)

Figure 2.2: Example of a sentence that has been annotated by an NLP stack.

To highlight ODIN’s capabilities, we show a simple example of an ODIN grammar in Listing 2.1. At runtime, ODIN applies this grammar over text that was previously annotated (like Figure 2.2) and records all text segments where rules match. We refer to these matches as *mentions*. The example grammar shown in Listing 2.1 operates as follows:

- The **ner** rule converts the BIO output of an external NER tool into ODIN entity mentions labeled **Protein**.

- The **ubiq** rule matches a ubiquitination event, which is anchored around a nominal predicate (**trigger**), “ubiquitination”, and has two arguments: a mandatory **theme**, which is syntactically attached to the verbal trigger through the preposition “of”, and an optional **cause**, attached to the trigger through the preposition “by”. Unlike entity mentions, ODIN event mentions keep track of their participants such as themes and causes, in addition of the matching text. The resulting event mention is assigned the **Ubiquitination** label. The hypernym labels **Simple_event** and **Event** are added automatically, according to the provided taxonomy.

- The **negreg** rule implements a negative regulation driven by a verbal predicate. Note that one of the arguments is an event produced by the **ubiq** rule.
Listing 2.1: Rules that capture the events listed in Figure 2.1.

Explicit priorities can be assigned to rules to control the order and extent of their execution. It is important to note that these priorities are not mandatory. If they are not specified, ODIN attempts to match all rules, which imposes an implicit execution. That is, ubiq can only match after ner is executed, because it requires entity mentions as arguments. Similarly, negreg matches only after the ubiquitination mentions are constructed.

Figure 2.3 shows the result of applying the grammar shown in Listing 2.1 over the sentence shown in Figure 2.2. The text in the transparent boxes indicate ODIN out-
put mentions. Nested mentions have arguments (e.g., Theme or Cause) which are represented as arrows pointing to other mentions.

![Figure 2.3: ODIN output mentions.](image)

This simple example highlights several of ODIN’s advantages. First, ODIN easily captures nested events, such as the negative regulations in the figure, simply by requiring an event’s argument to be another event (the negreg rule). Second, this example highlights one of the advantages of using syntax: because the ubiq rule in Listing 2.1 uses the prep_of relation to capture participants in the ubiquitination reaction, ODIN transparently captures the fact that there are two ubiquitination reactions in the sentence in Figure 2.2, which would be cumbersome to capture using surface patterns.

Of course, this simple example does not cover all of ODIN’s features. In the following section, we will describe the different features that can be used to make more general, permissive, or restrictive rules using our declarative language.

### 2.3 Rule Language

As the previous example illustrated, the fundamental building block of an ODIN grammar is a rule. Rules define either surface patterns, which are flat patterns over sequences of words, such as ner in the example (formally defined in Section 2.3.3), or patterns over the underlying syntactic structure of a sentence described using relational dependencies, such as ubiq, or negreg (defined in Section 2.3.4).

All ODIN grammars are written in YAML, which is a human-readable data serialization language (Ben-Kiki et al., 2005). However, it is not necessary to be a YAML expert to use ODIN, as we only use a small and simple YAML subset to write rules. A brief explanation of the required YAML features is given in Section 2.3.1.
Once you are comfortable writing rules, it is time to construct a complete domain grammar. In the simplest instance, a complete grammar is a single file containing some rules (similar to Listing 2.1). While this is sufficient for simple domains, when tackling more complex domains it may become necessary to organize rules into several files and recycle sets of prototypical rules to cover related events by altering sub-pattern variables. We describe all these situations in Section 2.3.5.

2.3.1 A Gentle Introduction to YAML

ODIN grammars are written using a small YAML subset. Other options like JSON and XML where considered, but YAML was chosen due to its readability. In particular, we only use lists, associative arrays, and strings, which are briefly summarized below. For more details (although you should not need them), please read the YAML manual (Ben-Kiki et al., 2005).

YAML Lists

YAML supports two different ways of specifying lists. The recommended one for ODIN requires each list item to appear in a line by itself, and it is denoted by prepending a dash and a space before the actual element. Elements of the same list must have the same level of indentation. As an example, a list of fruits in YAML notation is provided in Listing 2.2.

```
1 - apple
2 - banana
3 - orange
4 - watermelon
```

Listing 2.2: Example YAML list

YAML Associative Arrays

YAML supports two different syntaxes for associative arrays. The recommended one for ODIN is the one in which each key-value pair appears in its own line, and
all key-value pairs have the same level of indentation. Each key must be followed by colon. An example of a YAML associative array is provided in Listing 2.3.

```
1 first_name: Homer
2 last_name: Simpson
3 address: 742 Evergreen Terrace
4 town: Springfield
```

Listing 2.3: Example YAML associative array

**YAML Strings**

Many rule components are encoded using single-line strings, as we have seen in the previous examples. There is one exception: the rule’s pattern field (as described in Sections 2.3.3 and 2.3.4). Patterns can be complex and it is a good idea to break them into several lines. YAML supports multi-line strings using the vertical bar character (i.e., |) to partition a key-value pair. When this is used, the string begins in the next line and it is delimited by its indentation. An example of a YAML multi-line string is shown in Listing 2.4.

```
1 var1: single-line string
2 var2: |
3 this is a multi-line string
4 this is still part of the same string
5 because of its indentation
6 var3: another single-line string
```

Listing 2.4: Example YAML associative array with one multi-line string value

As shown, YAML strings don’t have to be quoted. This is a nice feature that allows one to write shorter and cleaner rules. However, there is one exception that you should be aware of: strings that start with a YAML indicator character must be quoted. Indicator characters have special semantics and must be quoted if they should be interpreted as part of a string. All valid YAML indicator characters are shown in Figure 2.4.
As you can probably tell, these are not characters that occur frequently in practice. Usually names and labels are composed of alphanumeric characters and the occasional underscore, so, most of the time, you can use unquoted strings.

2.3.2 Rules

ODIN rules are represented simply as YAML associative arrays, using the fields shown in Table 2.1.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>names</td>
<td>The rule’s name (must be unique)</td>
<td>must be provided</td>
</tr>
<tr>
<td>label</td>
<td>The label or list of labels to assign to the mentions found by this rule</td>
<td>must be provided</td>
</tr>
<tr>
<td>priority</td>
<td>The iterations in which this rule should be applied. Note that the ODIN runtime system continuously applies the given grammar on a given sentence until no new rule matches (this allows grammars that use recursive events, such as the one in Listing 2.1 to work). Each of these distinct runs are called “iterations”, and they are all numbered starting from 1. Through priorities, a developer can specify in which iteration(s) the corresponding rule should run. Specifying priorities is not required, but it may have an impact on run time, by optimizing which rule should be applied when. A priority can be exact (denoted by a single number), a range (two numbers separated by a dash), an infinite range (a number followed by a plus +), or a list of priorities (a comma separated list of numbers surrounded by square brackets).</td>
<td>1+</td>
</tr>
<tr>
<td>action</td>
<td>The custom code (or “action”) to call for the matched mentions. As discussed in Section ??, specifying an action is not required. The default action does the most widely used job, i.e., keeping track of what was matched.</td>
<td>default</td>
</tr>
<tr>
<td>keep</td>
<td>Include the output of this rule in the output results?</td>
<td>true</td>
</tr>
<tr>
<td>type</td>
<td>What type of rule is this: surface rule (token) or syntax-based (dependency)?</td>
<td>dependency</td>
</tr>
<tr>
<td>unit</td>
<td>As discussed in Section 2.3.3, each token contains multiple pieces of information, e.g., the actual word (word), its lemma, or its part-of-speech (POS) tag (tag). This parameter indicates which of these fields to be matched against implicitly, i.e., when the token pattern is a simple string. Currently, the only valid values are word and tag.</td>
<td>word</td>
</tr>
<tr>
<td>pattern</td>
<td>Either a token or a dependency pattern, as specified in type, that describes how to match mentions.</td>
<td>must be provided</td>
</tr>
</tbody>
</table>

Table 2.1: Overview of the fields of an ODIN rule.
Clearly, the most important part of a rule, is the pattern field. In Section 2.3.3 we describe how to implement surface, or “token”, patterns. These are useful for simple sequences, or when syntax is not to be trusted. In Section 2.3.4 we introduce the bread-and-butter of ODIN: syntactic, or “dependency”, patterns. Note that both types of patterns use some of the same constructs: string matchers (i.e., objects that can match a string), and token constraints (i.e., objects that impose complex conditions on individual tokens to be matched). We will introduce these for token patterns, and reuse them for dependency patterns.

2.3.3 Token Patterns

A common task in information extraction is extracting structured information from text. Structured information may refer to different kinds of things, from item enumerations to complex event mentions. One way to extract this kind of mentions from text is by the use of surface patterns that allow us to match sequences of tokens that usually signal the presence of the information we are interested in.

Surface patterns are available in ODIN through the use of “token” patterns. ODIN’s token patterns can match continuous and discontinuous token sequences by applying linguistic constraints on each token (Section 2.3.3), imposing structure (Section 2.3.3), generalized through the use of operators (Section 2.3.3), and drawing on context (Section 2.3.3). In this section we will describe each of these features that make token patterns efficient and easy to use for the different information extraction tasks that are encountered by practitioners.

Token Constraints

Remember that, in the simplest case, a token (or word) can be matched in ODIN simply by specifying a string. For example, to match the phosphorylation trigger in Listing 2.1, all we had to do was write phosphorylation (quotes are optional). But, of course, ODIN can do a lot more when matching individual words. This is where token constraints become useful. A token constraint is a boolean expression
surrounded by square brackets that can be used to impose more complex conditions when matching a token.

Each token has multiple fields that can be matched:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>The actual token.</td>
</tr>
<tr>
<td>lemma</td>
<td>The lemma form of the token.</td>
</tr>
<tr>
<td>tag</td>
<td>The part-of-speech (PoS) tag assigned to the token</td>
</tr>
<tr>
<td>incoming</td>
<td>Incoming relations from the dependency graph for the token</td>
</tr>
<tr>
<td>outgoing</td>
<td>Outgoing relations from the dependency graph for the token</td>
</tr>
<tr>
<td>chunk</td>
<td>The shallow constituent type (ex. NP, VP) immediately containing the token</td>
</tr>
<tr>
<td>entity</td>
<td>The NER label of the token</td>
</tr>
<tr>
<td>mention</td>
<td>The label of any Mention(s) (i.e., rule output) that contains the token</td>
</tr>
</tbody>
</table>

Table 2.2: An overview of the attributes that may be specified in a token constraint.

A token field is matched by writing the field name, followed by the equals character and a string matcher. (e.g. word=dog matches the word “dog”, tag=/^V/ matches any token with a part-of-speech that starts with “V”, entity="B-Person" matches any token that is the beginning of a person named entity). Expressions can be combined using the common boolean operators: and &, or |, not !. Parentheses are also available for grouping the boolean expressions.

Note: if the square brackets that delimit the token constraint are left empty, i.e., [], the expression will match any token.

String Matchers

A string matcher is an object that matches a string. Matching strings is the most common operation in ODIN, being heavily used both in token and dependency patterns. This is because all token fields (described in Table 2.2) have string values that are matched using string matchers. Additionally, dependency patterns (described
in Section 2.3.4) match incoming and outgoing dependencies by matching the name of the dependency using the same string matchers.

Strings can be matched exactly or using regular expressions. Both options are described next.

**Exact String Matchers**

An exact string matcher is denoted using a string literal, which is a single- or double-quote delimited string. The escape character is the backslash (e.g., \\). If the string is a valid Java identifier, the quotes can be omitted. For example, `word=dog` matches the word “dog”.

**Regex String Matchers**

A regex string matcher is denoted by a slash delimited Java regular expression. A slash can be escaped using a backslash. This is the only escaping done by ODIN to regular expressions, everything else is handled by the Java regular expression engine. For example, `tag=/^V/` matches any token with a part-of-speech that starts with “V”.

**Named Arguments**

Token patterns support two types of named arguments: those constructed “on-the-fly” from an arbitrary sequence of tokens or those that point to existing mentions.

Capturing a sequence of tokens and assigning a label to the span for later use can be performed using the `(?<identifier> pattern)` notation, where `identifier` is the argument name and `pattern` is the token pattern whose result should be captured and associated with the argument name. Capturing several sequences or mentions with the same name is supported as well as nested captures (i.e., arguments defined inside other arguments).

---

9See [http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html](http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html)
Bonnie and Clyde robbed the bank.

Bonnie and Clyde robbed the bank.

Listing 2.5: An example of a token pattern with a repeated argument using a subpattern-style named argument.

While powerful, these subpattern-style named arguments can quickly clutter a rule, especially when the pattern is nontrivial. Consider the (\(\text{?< robber }>\)) pattern in Listing 2.5. A broad-coverage rule for detecting a robber could be quite complex. A better strategy might be to generalize this pattern as a rule designed to identify any person. Since this rule provides the context of a robbery event, it would be sufficient to simply specify that the span of text being labelled robber is a mention of a person. We can do this quite easily with ODIN.

A previously matched mention can be included in a token pattern using the @ operator followed by a StringMatcher that should match a mention label. This will consume all the tokens that are part of the matched mention. If the mention should be captured in one of the named groups then the notation is @identifier:StringMatcher where the identifier is the group name and the string matcher should match the mention label.

Bonnie and Clyde robbed the bank.

Listing 2.6: An example of a token pattern with a repeated argument using an mention-based named argument. This assumes that other rules built the Person and Location mentions, possibly from the output of a NER.

Token Pattern Operations

The most fundamental token pattern operations are concatenation and alternation. Concatenating two patterns is achieved by writing one pattern after the other. Alternation is achieved by separating the two patterns using the alternation operator (e.g., |). This is analogous to a boolean OR.
Parentheses can be used to group such expressions. As is usual, parentheses take precedence over the alternation operator. Table 2.3 shows some simple examples of operator and parenthesis usage.

<table>
<thead>
<tr>
<th>pattern</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fat rats</td>
<td>mice</td>
</tr>
<tr>
<td>fat (rats</td>
<td>mice)</td>
</tr>
</tbody>
</table>

Table 2.3: Example of parentheses usage to change operator precedence.

ODIN also supports several types of quantifiers (see Table 2.4 for details). The ?, *, and + postfix quantifiers are used to match a pattern zero or one times, zero or more times, and one or more times respectively. These are greedy quantifiers, and can be turned lazy by appending a question mark (e.g., ??, *?, +?). Figure 2.5 illustrates the difference between greedy and lazy quantifiers.

<table>
<thead>
<tr>
<th>pattern</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a] +  c</td>
<td>a b c d e f c</td>
</tr>
<tr>
<td>[a] +? c</td>
<td>a b c</td>
</tr>
</tbody>
</table>

Figure 2.5: Comparison of greedy (default behavior) and lazy (?) quantifiers.

Ranged repetitions can be specified by appending \{n,m\} to a pattern, which means that the pattern should repeat at least n times and at most m. If n is omitted (e.g., \{,m\}) then the pattern must repeat zero to m times. If m is omitted (e.g., \{n,\}) then the pattern must repeat at least n times. Ranges are greedy, and can be turned lazy by appending a question mark (e.g., \{n,m\}?, \{,m\}? , \{n,\}??) For an exact number of repetitions the \{n\} suffix is provided. Since this is an exact repetition there are no greedy/lazy variations.

Table 2.4 summarizes this set of quantifiers.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Lazy form</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>The quantified pattern is optional.</td>
<td>??</td>
</tr>
<tr>
<td>*</td>
<td>Repeat the quantified pattern zero or more times.</td>
<td>*?</td>
</tr>
<tr>
<td>+</td>
<td>Repeat the quantified pattern one or more times.</td>
<td>+?</td>
</tr>
<tr>
<td>{n}</td>
<td>Exact repetition. Repeat the quantified pattern n times.</td>
<td></td>
</tr>
<tr>
<td>{n,m}</td>
<td>Ranged repetition. Repeat the quantified pattern between n and m times, where n &lt; m.</td>
<td>{n,m}?</td>
</tr>
<tr>
<td>{,m}</td>
<td>Open start ranged repetition. Repeat the quantified pattern between 0 and m times, where m &gt; 0.</td>
<td>{,m}?</td>
</tr>
<tr>
<td>{n,}</td>
<td>Open end ranged repetition. Repeat the quantified pattern at least n times, where n &gt; 0.</td>
<td>{n,}?</td>
</tr>
</tbody>
</table>

Table 2.4: An overview of the quantifiers supported by ODIN’s token patterns.

Quantifiers apply either to a single token constraint or to a group of token constraints. Groups are specified by using parentheses. An example of a token pattern that uses quantifiers is shown on Listing 2.7. This example also shows that one can use mention captures in the quantified groups (the `Number` argument), and that the captured mentions can share the same name. This is useful for the extraction of enumerations of unknown length.
The numbers 4, 8, 15, 16, 23 and 42 frequently recurred in Lost.

# First, find numbers by inspecting the POS tag.
# Note that this is not the only way to check for a number,
# there are other options, such as `[word=/\d+/]`
- name: numbers
  label: Number
  priority: 1
  type: token
  pattern: |
  [tag=CD]

# Second, match comma separated lists of numbers optionally
followed by the word 'and' and a final number.
- name: list
  label: ListOfNumbers
  priority: 2
  type: token
  pattern: |
  @num:Number ("," @num:Number)+ (","? and @num:Number)?

Listing 2.7: Example showcasing quantifiers and mention captures.

Zero-width Assertions

Zero-width assertions allow one to verify whether or not a pattern is present without including it in the matched result. ODIN supports the following zero-width assertions:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>beginning of sentence</td>
</tr>
<tr>
<td>$</td>
<td>end of sentence</td>
</tr>
<tr>
<td>(?=...)</td>
<td>positive lookahead</td>
</tr>
<tr>
<td>(!=...)</td>
<td>negative lookahead</td>
</tr>
<tr>
<td>(?&lt;=...)</td>
<td>positive lookbehind</td>
</tr>
<tr>
<td>(?&lt;!...)</td>
<td>negative lookbehind</td>
</tr>
</tbody>
</table>

Table 2.5: An overview of the zero-width assertions supported by ODIN. These patterns do not consume tokens, but are useful to match patterns preceding/following the expression of interest.
Output

The output of any ODIN rule is called a “mention”, and they are actual instances of a `Mention` Scala class, or one of its subclasses (see Section 2.4).

The inclusion of named captures in a token pattern affects the type of `Mention` that is produced. In general, the result of applying a token pattern successfully is usually a `TextBoundMention` (see Section 2.4). However, if the token pattern includes named captures, then the result is a `RelationMention`, which is essentially a collection of named captures, or “arguments” (but without a predicate, or “trigger”, which is typical of event mentions!). In other words, relation mentions are not dependent on a particular predicate. If one of the named captures has the name “trigger” (case insensitive), then ODIN assumes that this pattern defines an event, and the result is an event mention (an instance of the `EventMention` class). Listings 2.8 and 2.9 show two simple patterns that produce an event mention and a relation mention, respectively.

Oscar lives in a trash can.

```
- name: event_mention_out
  label: LivesIn
  priority: 2
  type: token
  pattern: |
    (?<resident>Oscar)
    (?<trigger>[lemma=live])
    in [tag=DT]? (?<location>[tag=/>N/]+)
```

Listing 2.8: An example of a token pattern rule that produces an event mention through the specification of a trigger.
Dr. Frankenstein spends a lot of time in the graveyard.

Listing 2.9: An example of a token pattern rule that produces a relation mention. This rule has named arguments, but does not specify a trigger. For brevity, we assume that Person mentions have already been identified.

2.3.4 Dependency Patterns

While token patterns are quite powerful, they are, of course, not too robust to syntactic variation. Writing patterns over syntactic structure produces generalizations with broader coverage that do not sacrifice precision. Consider the sentences in Figure 2.6:

Noam danced at midnight with the leprechaun.

Noam, in full view of the three-legged robot, danced at dawn with the leprechaun.

Noam danced under the moonlight at midnight with the leprechaun.

His friends watched in awe while Noam danced the forbidden jig with the leprechaun at midnight.

Figure 2.6: These sentences show some of the infinite syntactic variation describing a dance between two entities.
Figure 2.7: The relational-dependency parse for the sentences in Figure 2.6.

While it requires several token-pattern rules to precisely capture the syntactic variation shown in Figure 2.6, all of these variants can be covered with a single rule using a dependency pattern (see Listing 2.10).

Listing 2.10: A dependency rule that expects two arguments: (1) a nominal subject and (2) the head word complements of a “with” prepositional phrase off of the lemmatized trigger, dance; (2) may be preceded by an optional hop through a direct object (dobj) relation. Note the optional hop through a direct object (dobj). Parsers often struggle with prepositional attachment, so we have added an optional dobj in this rule to be robust to such errors.
Formally, a dependency pattern describes a traversal over a syntactic dependency graph. Again, we currently use Stanford dependencies de Marneffe and Manning (2008) in ODIN, but ODIN is independent of the representation used. ODIN’s dependency patterns are composed of several fields. Dependency patterns defining event rules require a “trigger” that must be set to a token pattern (see previous section). This token pattern describes a valid predicate for the event of interest. The rest of the fields are event arguments defined through a syntactic path from the trigger to some mention (entity or event) that was previously matched by another rule. The path is composed of hops and optional filters. The hops are edges in the syntactic dependency graph; the filters are token constraints on the nodes (tokens) in the graph.

Hops can be incoming or outgoing. An outgoing hop follows the direction of the edge from HEAD $\rightarrow$ DEPENDENT; an incoming hop goes against the direction of the edge, leading from DEPENDENT $\rightarrow$ HEAD. For example, the dependency “jumped” $\rightarrow$ “Fonzie” is outgoing (“jumped” is the head), but it is considered incoming when traversed in the other direction: “Fonzie” $\leftarrow$ “jumped”.

Figure 2.8: The structured output of the rule in Listing 2.10.

Figure 2.9: A simple sentence with its corresponding dependency parse tree.
Listing 2.11: A simple, two-argument dependency pattern composed solely of outgoing hops, which matches the “jumping” event above. We are assuming that a different rule created a Noun mention for every NN*.

An outgoing dependency is matched using the > operator followed by a string matcher, which operates on the label of the corresponding dependency, e.g., >nsubj. Because most patterns use outgoing hops, (i.e. HEAD → DEPENDENT), the > operator is implicit and can therefore be omitted. An incoming relation (i.e. DEPENDENT → HEAD) is matched using a required < operator followed by a string matcher. > is a wildcard operator that can be used to match any outgoing dependency. < is a wildcard operator that can be used to match any incoming dependency.

Importantly, restrictions may be imposed on the nodes (i.e., tokens) visited when following dependencies, using the usual token constraints. Listing 2.12 illustrates such constraints on both the robber (using the POS tag) and the property (using the actual word).
Gonzo stole her chicken.

Gonzo stole her heart.

Listing 2.12: While these two sentences are syntactically identical, only one pertains to theft of tangible goods. We are assuming that a different rule created a Noun mention for every NN*.

Just as in token patterns, dependency patterns can include parentheses and the alternation operator |. For example, the pattern nsubj|agent matches an outgoing dependency whose label is either nsubj or agent.

Named Arguments for Dependency Patterns

Clearly, naming event arguments is important (e.g., one may want to keep track who is the agent and who is the patient in a robbery event). We probably already observed that ODIN has a simple syntax for this: a path to an argument begins with name:label = path, where label is the the label of an existing Mention. The path must lead to a token that is a part of a Mention with the specified label. Argument names are required and unique, i.e., you can’t have two different patterns with the same name. But the same pattern may match multiple mentions! If, for example, an argument with the name “theme” matched three different entities, then three event
mentions will be generated, each with one entity as the theme. If the given path to the theme fails to match any Mention, then no event mentions will be created.

At times one may need to make an argument optional or allow for more than one argument with the same name in a single event mention. This can be achieved through the use of argument quantifiers. Arguments can be made optional with the ? operator. The + operator is used to indicate that a single event mention with all matches should be created. The * is similar to + but also makes the argument optional. If the exact number of arguments with the same name is known, it can be specified using the exact repetition quantifier \{k\}. In cases of exact repetitions, the cartesian product will be applied to the Mentions matching the given path. If \( k \) Mentions are asked for in a path \( p \) and \( n \) are found to match \( p \), then \( j \) event mentions will be produced, where \( j \) is the binomial coefficient shown in Equation 2.3.4.

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]  

Figure 2.10: The sentence used by the rules shown in Listings 2.13, 2.14, & 2.15. We are assuming that a different rule created a Location mention for every location NE.

| - name: cities-1 |
| label: Cities |
| priority: 2 |
| pattern: |
| trigger = [lemma=city] |
| # produces 4 EventMentions each with 1 city |
| city:Location = prep_like |

Listing 2.13: An example of a complete dependency pattern rule without an argument quantifier.
Table 2.6: The four mentions produced by the dependency pattern shown in Listing 2.13.

<table>
<thead>
<tr>
<th>Mention</th>
<th>Cities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>London</td>
</tr>
<tr>
<td>2</td>
<td>Paris</td>
</tr>
<tr>
<td>3</td>
<td>Tokyo</td>
</tr>
<tr>
<td>4</td>
<td>Beijing</td>
</tr>
</tbody>
</table>

Listing 2.14: An example of a dependency pattern with a + argument quantifier. Its output is shown in Table 2.7.

<table>
<thead>
<tr>
<th>Mention</th>
<th>Cities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>London, Paris, Tokyo, Beijing</td>
</tr>
</tbody>
</table>

Table 2.7: The single mention produced by the rule shown in Listing 2.14.

Listing 2.15: An example of a dependency pattern with a \{k\} quantifiers on event arguments. The scattering effect of the \{k\} quantifier is shown in Table 2.8.

<table>
<thead>
<tr>
<th>Mention</th>
<th>Cities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>London, Paris</td>
</tr>
<tr>
<td>2</td>
<td>London, Tokyo</td>
</tr>
<tr>
<td>3</td>
<td>London, Beijing</td>
</tr>
<tr>
<td>4</td>
<td>Paris, Tokyo</td>
</tr>
<tr>
<td>5</td>
<td>Paris, Beijing</td>
</tr>
<tr>
<td>6</td>
<td>Tokyo, Beijing</td>
</tr>
</tbody>
</table>

Table 2.8: The six mentions produced by the rule shown in Listing 2.15.
Quantifiers for Dependency Patterns

In addition of the above quantifiers on event arguments, ODIN supports quantifiers inside the actual dependency patterns. They are shown in Table 2.9.

The ?, * and + postfix quantifiers are used to match a pattern zero or one times, zero or more times, and one or more times respectively. There is no notion of greedy/lazy dependency patterns.

Ranged repetitions can be specified by appending \{m,n\} to a pattern, and means that the pattern should repeat at least m times and at most n. If m is omitted (e.g., \{,n\}) then the pattern must repeat zero to n times. If n is omitted (e.g., \{m,\}) then the pattern must repeat at least m times. There is no notion of greedy/lazy dependency patterns. For an exact number of repetitions the \{n\} suffix is provided.

For example, the pattern /prep/+ matches a sequence of 1 or more outgoing dependencies whose labels contain prep. The pattern dobj* /prep/{,3} matches 0 or more dobj dependencies, followed by up to 3 outgoing dependencies that contain prep.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>The quantified pattern is optional.</td>
</tr>
<tr>
<td>*</td>
<td>Repeat the quantified pattern zero or more times.</td>
</tr>
<tr>
<td>+</td>
<td>Repeat the quantified pattern one or more times.</td>
</tr>
<tr>
<td>{n}</td>
<td>Exact repetition. Repeat the quantified pattern n times.</td>
</tr>
<tr>
<td>{n,m}</td>
<td>Ranged repetition. Repeat the quantified pattern between n and m times, where n &lt; m.</td>
</tr>
<tr>
<td>{,m}</td>
<td>Open start ranged repetition. Repeat the quantified pattern between 0 and m times, where m &gt; 0.</td>
</tr>
<tr>
<td>{n,}</td>
<td>Open end ranged repetition. Repeat the quantified pattern at least n times, where n &gt; 0.</td>
</tr>
</tbody>
</table>

Table 2.9: An overview of the quantifiers supported by ODIN’s dependency patterns.
Zero-width Assertions

For dependency patterns, there no lookbehind or lookahead assertions, only lookarround assertions. The lookarround syntax is (?= pattern) for positive assertions and (!? pattern) for negative assertions. Listing 2.16 shows an example of a positive lookarround in action.

Dennis crashed his mom’s car.

Dennis crashed his mom’s car.

Listing 2.16: Sometimes ownership matters. Perhaps we want to know whether or not Dennis should be grounded for crashing a car. Did Dennis crash his mother’s car? A positive lookarround is needed for this.

Output

The result of applying a dependency pattern successfully is usually an event mention. If a trigger is not specified, a relation mention is produced (see Figures 2.17 & 2.18 for details).
Oscar lives in a trash can.

Listing 2.17: An example of a dependency pattern rule that produces an event mention through the specification of a trigger.

Dr. Frankenstein spends a lot of time in the graveyard.

Listing 2.18: An example of a dependency pattern rule that produces a relation mention. This rule has named arguments, but does not specify a trigger. When the trigger field is omitted in a dependency pattern, the first field given should specify a named argument using the mention retrieval syntax (argname:MentionLabel). All subsequent dependency patterns used by the other arguments are anchored on this first argument.

2.3.5 Building a Grammar

By now, we hope you are somewhat confident that you can write ODIN rules. Of course, the next step is to put them together into a complete grammar. This can be very simple: minimally, all you have to do is to store them all into a single file which is then loaded into an ODIN engine (see Section ??). If you care a lot about efficiency, you can tune your grammar by assigning priorities to rules. For example,
rules that match entities should be executed before (i.e., have a lower priority) than rules that match events where these entities serve as arguments. (But again, this is not needed: ODIN takes care of pipelining rules internally.)

But some domain grammars are more complicated than a simple sequence of rules. You may have event labels that are so complex that you would prefer to store them in a taxonomy. Some event types have almost exactly the same syntactic representations as others, so you would like to reuse some rules. ODIN supports all these issues. We describe them next.

Master File

The master file is a grammar’s entry point, or the file is passed to the ODIN runtime engine. As mentioned, for simple grammars, this file can be simply a collection of rules. For more complicated scenarios, this file must contain a required rules section, and two optional sections: taxonomy and vars. Let us describe these sections next.

Taxonomy

The taxonomy is a forest (meaning a collection of trees) of labels that, if specified, is used by ODIN as the hierarchy for mention labels. An example taxonomy is shown in Listing 2.19.
# a tree hierarchy can be used to define the taxonomy
- organism:
  - prokaryotic:
    - archaeabacteria
    - eubacteria
  - eukaryotic:
    - unicellular:
      - protista
    - multicellular:
      - autotrophic:
      - heterotrophic:
        - fungi
        - animalia

# we want to include robots in our taxonomy
# but they are not organisms, what can we do?
# fortunately, multiple trees are supported
- robot

Listing 2.19: Example taxonomy

If a taxonomy is provided, then all the labels used by the rules must be declared in the taxonomy. This is obviously useful for catching typos. More importantly, the taxonomy hierarchy is used to robustly match mentions in subsequent rules. For example, if a rule creates an entity mention with the label `animalia` from the taxonomy in Listing 2.19, this mention will be matched as argument in a subsequent rule, which requires that argument to be of label `multicellular`. This is because `animalia` is a hyponym of `multicellular`, i.e., it is directly derived from it.

If the value of `taxonomy` is a single string, then it will be interpreted as a file name and the taxonomy will be read from that file. It should be noted that the taxonomy may only be specified in the master file, whether included directly or provided through an import (see Listing 2.20).

```yaml
# the taxonomy file should contain only the contents of the taxonomy (without the taxonomy: section name)
taxonomy: path/to/taxonomy.yml
```

Listing 2.20: An example of a taxonomy import.
Variables and Templates

It is very common that similar events share the same syntactic structure. For example, in the biomedical domain, all the biochemical reactions (there are between 10 and 20 of these) share the same structure. For example, “A phosphorylates B” is similar to “A ubiquitinates B”, with the exception of the predicate: “phosphorylates” vs. “ubiquitinates”. In such situations, we would like to reuse these syntactic structures between events (so we do not write the same rules 10–20 times). ODIN supports these through the use of variables and rule templates, where rule templates are simply rules that use variables. For example, one could write a single rule template for the above example, where the trigger constraints are encoded using a variable.

In general, variables can be declared as a YAML mapping, and they can be substituted in rules using the `${variableName}` notation (see Examples 2.21 & 2.22). Furthermore, wherever a rule can be specified, you can also import a file, through the `import` command, and its optional `vars` parameter. This gives one a further opportunity to instantiate variables. Listing 2.21 shows the `import` command in action.
Listing 2.21: An example of a master file that uses import statements and demonstrates variable precedence. Note that variables can be instantiated in three different places: (a) in the template file itself, (b) when the import command is used, or (c) at the top of the master file. The precedence is: (b) > (c) > (a). For this particular example, it means that the value chosen for the myTrigger variable is “eat” for the first import (LINE 9) and “sell” for the second import (LINE 13).
vars:
  # these variables are superseded by those in the master file
  myTrigger: buy
  myMentionLabel: Food

rules:
- name: example_rule
  label: Event
  type: token
  priority: 1
  pattern: |
    @person:Person  # match a person
    (?<trigger> [lemma=${myTrigger}])  # trigger comes from provided variable
    [tag="DT"]? @object:${myMentionLabel}  # retrieve mention with given label

Listing 2.22: The template.yml file imported in Listing 2.21.

2.4 Mentions, or the Output of Rules

As hinted before in this document, each rule produces a Mention object when it successfully matches some text. These objects are nothing magical: we simply use them to store everything that the rule contains, and the corresponding text matched. Table 2.10 summarizes the fields of the mention object.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>labels</td>
<td>The sequence of labels to associate with a mention.</td>
</tr>
<tr>
<td>tokenInterval</td>
<td>The open interval token span from the first word to the final word +1.</td>
</tr>
<tr>
<td>startOffset</td>
<td>The character index in the original text at which the mention begins.</td>
</tr>
<tr>
<td>endOffset</td>
<td>The character index in the original text at which the mention ends +1.</td>
</tr>
<tr>
<td>sentence</td>
<td>The sentence index of this mention.</td>
</tr>
<tr>
<td>document</td>
<td>The document (composed of annotated sentences) from which this mention originates.</td>
</tr>
<tr>
<td>arguments</td>
<td>A map from argument name (a String) to a sequence of mentions.</td>
</tr>
<tr>
<td>foundBy</td>
<td>The name of the rule that “found” this mention.</td>
</tr>
</tbody>
</table>

Table 2.10: An overview of the most important fields of the Mention class.
Sometimes, actual code is best at explaining things. We highly encourage the reader to take a look at the code implementing `Mention` and its subclasses\textsuperscript{10}. Note that some the information stored in mentions, e.g., the token interval of the mention, refer to data structures produced by our preprocessing code, such as `Sentence` and `Document`. Again, reading through the code is the best way to learn about these\textsuperscript{11}.

2.4.1 TextBoundMention

The `Mention` class is subclassed by several other classes. The simplest is `TextBoundMention`. A `TextBoundMention` is created when the output of a rule is a flat structure, i.e., a contiguous sequence of tokens in a sentence. More formally, a `TextBoundMention` will have a `tokenInterval`, but its `arguments` map will be empty. These mentions are usually used to represent entities or event triggers.

2.4.2 RelationMention

A `RelationMention` encodes $n$-ary relations between its arguments. All the arguments are named (based on the argument names specified in the matched rule), and are stored in the `arguments` map. Importantly, several arguments may have the same name! This is extremely useful when one needs to capture in a rule enumerations of several valid arguments with the same role (for example, a `food` argument may capture multiple foods consumed at a dinner).

2.4.3 EventMention

An `EventMention` is similar to a `RelationMention`, with just one additional feature: it has a `TextBoundMention` that represents the trigger of the event. In other words, the `arguments` map contains an additional argument, labeled `trigger`, which stores the event’s predicate. Note: an event must have exactly one trigger.

\textsuperscript{10}https://github.com/clulab/processors/blob/master/src/main/scala/edu/arizona/sista/odin/Mention.scala

CHAPTER 3

IMPLEMENTATION OF THE INFORMATION EXTRACTION FRAMEWORK

3.1 Overview

The information extraction framework described in this chapter is designed to handle the complexity of natural language by extracting information and representing it in data structures that can potentially be nested. This is done by applying a cascade of finite-state automata (following the approach first proposed by Appelt et al. (1993)) to the input text which returns a collection of matches, such as names of people, locations, organizations, and the relations between them. These are implemented as ODIN Mentions, which are described in Section 2.4.

ODIN is designed to apply the cascade of automata over arbitrary text. Even more so, it needs to do it rapidly, for practical concerns to be able to operate over large document collections. A whole cascade is defined declaratively as a collection of rules, which we refer to as a grammar. Each rule in the grammar represents one of the automata that comprise the cascade. ODIN compiles each of the rules in the grammar into an Extractor object. The collection of Extractors corresponding to the cascade is managed by an ExtractorEngine, which is responsible for applying the Extractor objects in the correct order and for managing the resulting Mentions.

Following is a description of how these Extractors are constructed and applied over free text.

3.2 Extractor

As mentioned above, each automata that forms part of the cascade is encoded as an Extractor object. The role of an Extractor, as its name implies, is to scan a sentence for matching mentions, by following Algorithm 1. When an Extractor
scans a sentence, it has access to the annotated Document and to a State object that contains all the mentions found so far. State are described in Section 3.3.1. The resulting mentions are then passed to a function called action. This function returns a collection of mentions, which is then returned by the Extractor as the final result of the extraction.

**Algorithm 1:** Formal implementation of an Extractor, which finds mentions in a document conforming to a specified pattern.

```
Data: annotated document, state
Result: collection of mentions

mentions := ϵ;
for sent ← sentences(document) do
    ms := extract(sentence, document, state);
    append(mentions, ms);
end
return action(mentions, state);
```

Extractors are constructed by compiling a rule (please see Section 2.3 for a detailed explanation of ODIN rules). An Extractor gets several attributes from its corresponding rule, such as its name and priority. They also get a reference to the Action function specified in the rule. More importantly, each rule contains a pattern that defines the automata’s behavior.

One of ODIN’s main strengths is its ability to use automata of different types within the same cascade. This allows us to make use of a different Extractor type for each rule according to the rule’s requirements. Particularly, ODIN implements two Extractors that are useful for analyzing free text: a DependencyExtractor, which works over the sentence’s syntactic representation; and a TokenExtractor, which works over the sentence tokens. By using both types of Extractors in a single cascade, we can take advantage of the flexibility of syntax and the robustness of surface forms.

Following is an explanation of both kinds of Extractors.
3.2.1 Token Extractor

A TokenExtractor uses a token pattern to extract mentions from text. Token patterns are described in Section 3.4.1. If the pattern matches, the extractor examines the results and constructs a mention according to these rules:

- If the match has no captured mentions or intervals, then a TextBoundMention is constructed
- If one of the captures has the name “trigger” then an EventMention is created
- Otherwise, return a RelationMention

3.2.2 Dependency Extractor

DependencyExtractors find mentions by following paths in the dependency graph, as specified by the patterns expressed in the rule. Mentions returned by a DependencyExtractor are either EventMentions (if a trigger is provided) or RelationMentions (if there is no trigger). These patterns are described in Section 3.4.2.

3.2.3 Actions

An action is a function that is called an extractor to process the extracted mentions. The purpose of having an action function is to give the user an opportunity to intervene during the extraction process. This is desirable because ODIN is meant to be extensible and versatile. By using actions, hybrid systems can be built that combine our declarative rules with custom code that is not directly declarative. For example, actions could implement custom mention transformations, queries to an external database, logging of the found mentions during runtime, or any other need the user may have.

An action function takes two inputs: the extracted mentions and the current state. It returns a collection of mentions, which is then returned by the extractor as the final result of the extraction process, as shown in Algorithm 1.
All actions should be implemented in an Actions object, which is a regular Java or Scala object that inherits from the `Actions` trait. Actions can be implemented as Scala or Java functions, as shown in Listings 3.1 and 3.2.

```scala
1 def action(mentions: Seq[Mention], state: State): Seq[Mention]
```
Listing 3.1: Scala action signature.

```java
1 java.util.List<Mention> action(java.util.List<Mention> mentions, State state)
```
Listing 3.2: Java action signature.

The desired action is specified by name in the rule’s action field and it is retrieved from the actions object at runtime. This late-binding of the action function is achieved through reflection (Malenfant et al., 1996). The default action is the identity action, which returns the mentions unmodified.

### 3.3 Extractor Engine

The `ExtractorEngine` manages the extractors that comprise a single cascade. The `ExtractorEngine` is used to extract mentions from an annotated document by following Algorithm 2. The `ExtractorEngine` applies the whole cascade to the input text by ensuring that the `Extractors` execute in the correct order, and keeps track of the resulting mentions in a `State` object, which is available to subsequent automata. Once the cascade has completed the extraction, a `globalAction` function is called as a post-processing step. This `globalAction` is useful because it gives the opportunity to filter or modify the final results of the whole extraction process. Unlike the extractor’s actions that only have access to the extractor’s findings and the current state, the `globalAction` gets access to all the resulting mentions of the entire extraction process, which enables it to make global decisions.

Note that the `ExtractorEngine` holds no global state. Each call to Algorithm 2 is completely independent and therefore parallelizable. We take advantage of this feature in the REACH system, described in Chapter 4.
3.3.1 State

The State object holds an inverted index (Manning et al., 2008) that maps document positions to mentions. This inverted index can be used to retrieve mentions by position in an efficient manner. This is a common ODIN operation, primarily because the cascade of automata requires retrieving mentions by position each time an existing mention is required as part of its input. Also, users usually want to condition the creation of a new mention on its overlap with a previously found mention in the same position and with an incompatible label. An example of this situation is shown in Listing 3.3, where a rule used in the REACH system returns “Site” mentions only if they don’t overlap with previously found mentions with labels “Gene_or_gene_product” or “Simple_chemical”.

```
1 - name: site_1letter
2  label: Site  # this rule produces a "Site" mention
3  priority: 5  # it is applied at iteration 5
4  type: token  # it is a surface pattern
5  action: mkBioMention  # it makes use of the "mkBioMention" action
6  pattern: |
7    # ensure it is not preceded by the words "table" or "figure"
8    (?<![word=/(?i)"table|figure"/])
9    # match token if is a single letter followed by one or more digits
10   [word=/(?i)^[ACDEFGHIKLMNQRSTVWY]\d+$/
11     &  # AND
12   # the token should not overlap with a protein or chemical mention
13   !mention=/^(Gene_or_gene_product|Simple_chemical)$/]
```

Listing 3.3: Example REACH rule

Current extractors only allow the addition of mentions to the state, although nothing prevents the implementation of an extractor that would delete mentions. We chose not to delete mentions from the state because in our experience it was more useful to report all matched mentions and filter them as part of the application’s domain logic.
3.4 Patterns

ODIN supports two kinds of patterns: token and dependency patterns. The syntax for each pattern is described in Sections 2.3.3 and 2.3.4, respectively. The Backus-Naur Form grammars for both pattern languages are shown in Appendix A. In this section we will describe their implementation.

The parsers for both types of patterns are implemented using Scala’s parser combinators (Moors et al., 2008). Parser combinators are higher-order functions that take one or more parsers as input and return a single parser as output (Hutton, 1992). They facilitate the construction of recursive-descent parsers.

3.4.1 Token Pattern

A token pattern is meant to describe a pattern over a sequence of tokens. Each token can be annotated with several attributes as shown in Table 2.2.

Following Cox (2009), we implemented our token pattern automata as a bytecode that is executed by a virtual machine (VM). The VM simulates a non-deterministic finite-state automata (NFA) by keeping all the possible states of the automata at a given time in different “threads”. Each thread maintains its own state, which consists of: the next token to explore, the next instruction to execute, the captured mentions, the captured token intervals, and a stack with the current partial captures.

One advantage of the VM approach is that new features are easy to add by the addition and implementation of new VM instructions. Another advantage is that it does not use backtracking, which means it is not susceptible to pathological expressions (Cox, 2007).

Following is a description of how token patterns are compiled and evaluated.

Compilation

The token patterns are compiled into VM instructions by following Thompson’s construction algorithm (Thompson, 1968; Aho et al., 1986). The VM instructions are shown in Table 3.1.
Each instruction has a next pointer to the next instruction in the program, except for the Split instruction, which represents a bifurcation in the program execution and therefore has two following instructions associated with it.

To aid the construction of the NFA, we use a class called ProgramFragment that keeps track of one input and one or more outputs corresponding to each NFA fragment.

A concatenation of two ProgramFragments is implemented by simply setting the next pointers of all outputs of the first fragment to be the input of the second one. Thus, the resulting concatenated fragment has the input of the first as its input and the outputs of the second as its output.

An alternation is compiled by creating a new ProgramFragment whose input is a Split instruction that points to the two alternating fragments. The output of the resulting fragment is the concatenation of the alternating fragments’ output. A visualization of the resulting NFA is shown in Figure 3.1, and the VM instructions are shown in Listing 3.4.

```
1 in = Split(fragment1.in, fragment2.in)
2 out = fragment1.out ++ fragment2.out
3 ProgramFragment(in, out)
```

Listing 3.4: Pseudocode for the implementation of the composite alternation NFA shown in Figure 3.1
The Kleene star is also compiled by the use of a \texttt{Split} instruction. Note that ODIN supports two versions of the Kleene star: a greedy version, which tries to match as much as possible, and a lazy one, which tries to match as little as possible. Both are supported, by taking advantage of the fact that the first instruction of \texttt{Split} is always preferred over the second. A visualization of the resulting NFA is shown in Figure 3.2, and the VM instructions for the greedy and lazy versions of the Kleene star are shown in Listings 3.5 and 3.6 respectively.

![Figure 3.2: Composite NFA for the Kleene star.](image)

---

**Listing 3.5:** Pseudocode for the implementation of the Greedy Kleene star.

```plaintext
1 epsilon = Pass()
2 in = Split(fragment.in, epsilon)
3 fragment.out = in
4 out = epsilon
5 ProgramFragment(in, out)
```

**Listing 3.6:** Pseudocode for the implementation of the Lazy Kleene star.

```plaintext
1 epsilon = Pass()
2 in = Split(epsilon, fragment.in)
3 fragment.out = in
4 out = epsilon
5 ProgramFragment(in, out)
```

The optional operators (i.e., \( ? \) and \( ?? \)) are implemented by relying on the behavior of the \texttt{Split} instruction. Figure 3.3 shows a visualization of the NFA for
the optional operators and Listings 3.7 and 3.8 show their greedy and lazy bytecodes, which differ only in the parameter order of the \texttt{Split} instruction.

![Figure 3.3: Composite NFA representing optionality of a pattern.](image)

Figure 3.3: Composite NFA representing optionality of a pattern.

```plaintext
1 epsilon = Pass()
2 in = Split(fragment.in, epsilon)
3 out = fragment.out ++ epsilon
4 ProgramFragment(in, out)
```

Listing 3.7: Pseudocode for the implementation of the Greedy Optional NFA.

```plaintext
1 epsilon = Pass()
2 in = Split(epsilon, fragment.in)
3 out = fragment.out ++ epsilon
4 ProgramFragment(in, out)
```

Listing 3.8: Pseudocode for the implementation of the Lazy Optional NFA.

The Kleene plus construction, which matches a pattern one or more times, is shown in Figure 3.4, and the corresponding greedy and lazy bytecode is shown in Listings 3.9 and 3.10. Again, these differ only in the order of the parameters to the \texttt{Split} instruction.

![Figure 3.4: Composite NFA representing a pattern to be matched one or more times (Kleene plus).](image)

Figure 3.4: Composite NFA representing a pattern to be matched one or more times (Kleene plus).
Range quantifiers (shown in Table 2.4) are provided as syntactic sugar for specifying restricted repetitions. They are expanded into expressions involving the optional (i.e., ? and ??) and Kleene star (i.e., * and *?) operators. Examples of ranges and their corresponding expanded expressions are shown in Table 3.2.

<table>
<thead>
<tr>
<th>Range</th>
<th>Description</th>
<th>Expanded expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P{3})</td>
<td>Pattern (P) repeated exactly 3 times</td>
<td>(P \ P \ P)</td>
</tr>
<tr>
<td>(P{3,})</td>
<td>Pattern (P) repeated at least 3 times (greedily)</td>
<td>(P \ P \ P \ P^*)</td>
</tr>
<tr>
<td>(P{3,}?)</td>
<td>Pattern (P) repeated at least 3 times (lazily)</td>
<td>(P \ P \ P \ P^*?)</td>
</tr>
<tr>
<td>(P{,3})</td>
<td>Pattern (P) repeated at most 3 times (greedily)</td>
<td>(P^? \ P^? \ P^?)</td>
</tr>
<tr>
<td>(P{,3}?)</td>
<td>Pattern (P) repeated at most 3 times (lazily)</td>
<td>(P^?? \ P^?? \ P^??)</td>
</tr>
<tr>
<td>(P{3,6})</td>
<td>Pattern (P) repeated 3 to 6 times (greedily)</td>
<td>(P \ P \ P \ P^? \ P^? \ P^?)</td>
</tr>
<tr>
<td>(P{3,6}?)</td>
<td>Pattern (P) repeated 3 to 6 times (lazily)</td>
<td>(P \ P \ P \ P^? \ P^? \ P^? \ P^??)</td>
</tr>
</tbody>
</table>

Table 3.2: Explanation of range quantifier compiler expansions for token patterns.

**Execution**

As described previously, the VM simulates an NFA by representing all the simultaneous states as VM threads. Each thread holds all the data associated with it: the next token to explore, the next instruction to execute, the named captures
(both mentions and token intervals), and the token interval captures that have been partially captured.

This VM differs from the one proposed by Cox (2009) in several respects. First, our VM organizes threads differently in order to support mention overlap, which means that this VM can return several results for a single pattern if there are overlapping mentions in the State. Second, our VM supports reverse execution, which allows us to implement efficient unrestricted lookbehind assertions. This feature is not commonly implemented in other regex engines.¹ Third, because of the need to match mentions of different lengths (using the @ notation) and because the need to support lookaround assertions, we cannot run the threads in lock step, since they get out of sync once we start consuming mentions. Also, since our vocabulary is composed of boolean expressions over the token attributes, it is virtually infinite, which makes running the VM in lock step impractical.

New threads are spawned by the Split instruction, and they are ordered left-to-right by decreasing order of precedence. Once a thread reaches the Done instruction it is considered a successful match and all the threads to its right can be immediately dropped. All the threads to its left, however, still need to be evaluated, since they have higher precedence. Each thread can have at most one match.

In order to support overlapping mentions, we use thread bundles, which are sequences of threads encapsulated in a single object that acts as a regular thread. An example of a thread bundle is provided in Figure 3.5, which shows a sequence of threads with priorities 1 to 6 (one being the highest priority). Thread 4 is a thread bundle that is managing three sequences of threads, each with their own priorities. If any thread inside the thread 4 bundle matches successfully, then the bundle as a whole is considered a successful match and threads 5 and 6 can be dropped immediately. Each of the thread sequences inside the bundle can find a match, which means that a thread bundle can potentially have many matches, and all of them are reported if the bundle is the final match.

¹To the best of our knowledge, only .NET regex has the ability to evaluate unrestricted lookbehind assertions.
Figure 3.5: A sequence of VM threads where each thread represents a state of the NFA. Thread 4 represents a thread bundle, which is a nesting of several thread sequences.

One notable feature of the VM is its ability to execute instructions backwards (i.e., right-to-left) in the input text. This feature allows us to implement efficient unrestricted lookbehind assertions. This is useful, because lookarounds are used by our rules to restrict matches to certain contexts, which may depend on previously found mentions or other complex patterns. This also opens the door for other optimizations, which we discuss in Chapter 7.

Lookarounds are implemented by compiling the instructions of a pattern in a lookbehind in reverse order, and the instructions of a pattern in a lookahead in forward order. This allows the nesting of lookarounds. Each thread knows its own direction, and threads that form part of a lookbehind follow a right-to-left direction in the input text. This means that the lookbehind pattern only needs to be executed once, independent of the pattern’s complexity. Lookarounds are evaluated by a new VM instance. It is of note that the VM requires no initialization, so this operation is not expensive.

3.4.2 Dependency Pattern

Dependency patterns describe patterns over the dependency graph. They are defined as a series of paths in the graph, all of which are anchored on a mention, which could be either a preexisting mention or one created on-the-fly by the use of a trigger token pattern. Dependency patterns support two main types of operations: traversing the
graph by matching edge labels, and filtering tokens by evaluating token constraints. Their syntax is described in Section 2.3.4.

**Compilation**

Dependency patterns are composed of either a trigger token pattern or a label for matching existing mentions to be used as anchors. They also have a collection of `ArgumentPattern` objects that can find arguments by following a path in the dependency graph starting from the anchor. Each of these `ArgumentPatterns` is an automata over the dependency graph.

The `ArgumentPatterns` that describe the graph traversals are compiled into an abstract syntax tree (AST) in which each node represents an instruction to be evaluated. ASTs are constructed using ten different instructions, shown in Table 3.3.

The AST for each graph traversal is wrapped in a `ArgumentPattern` object that matches the AST starting at the anchor and groups the results according to the argument quantifiers, as described in Section 2.3.4.

Syntactic sugar is provided for the Kleene plus operator and the range quantifiers. Examples of quantifiers and their corresponding expanded expressions are shown in Table 3.4.

<table>
<thead>
<tr>
<th>Range</th>
<th>Description</th>
<th>Expanded expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>P+</td>
<td>Pattern P repeated one or more times</td>
<td>P P*</td>
</tr>
<tr>
<td>P{3}</td>
<td>Pattern P repeated exactly 3 times</td>
<td>P P P</td>
</tr>
<tr>
<td>P{3,}</td>
<td>Pattern P repeated at least 3 times</td>
<td>P P P P*</td>
</tr>
<tr>
<td>P{,3}</td>
<td>Pattern P repeated at most 3 times</td>
<td>P? P? P?</td>
</tr>
<tr>
<td>P{3,6}</td>
<td>Pattern P repeated 3 to 6 times</td>
<td>P P P P? P? P?</td>
</tr>
</tbody>
</table>

Table 3.4: Explanation of range quantifier compiler expansions for dependency patterns.

**Execution**

Execution of the dependency pattern is done by first matching either existing mentions with the specified label, or by applying token patterns that match mentions on-the-fly. This matched mention is used as an anchor for the argument extraction.
Execution of the AST is conducted directly by the AST itself. The AST may be positioned at many tokens at the same time, similarly to how token patterns work. This means that backtracking is not necessary. Once the tokens at the end of the path defined by the AST are retrieved, the ArgumentMatcher object looks for mentions in the State that are located at those positions and have the corresponding label. ArgumentMatcher objects also pack the results according to their argument quantifiers.

For example, Listing 3.11 shows a toy grammar with two rules. Rule attributes finds mentions of the words “ice” and “fire”. Rule firesong finds mentions of the word “song” that are attached through an “of” preposition to a word that begins with the letter “f” and is also a mention with label Attr. Figure 3.6 shows a sentence where this toy grammar can be applied. Note that rule firesong starts at the word “song” and then follows a prep_of dependency. At this point the Extractor has two simultaneous positions: the words “ice” and “fire”. Then, the token constraint that requires the word to start with the letter “f” is applied, which filters out the word “ice” and only the word “fire” remains. Lastly, the ArgumentPattern checks that the resulting word is part of a Attr mention, and if it does, then we obtain a successful match, as shown in Figure 3.6b.

```plaintext
1 - name: attributes
2   label: Attr
3   type: token
4     pattern: |
5       ice | fire
6
7 - name: firesong
8   label: FireSong
9   type: dependency
10  pattern: |
11    trigger = song
12    attribute: Attr = prep_of [word=/^-f/]  
```

Listing 3.11: Toy grammar with rules that find mentions of “song of fire”.
Figure 3.6: Example of an input sentence to the grammar shown in Listing 3.11.
Data: annotated document

Result: collection of mentions

\( state := \epsilon; \)

\( \text{for } i \leftarrow 1 \text{ to } \infty \text{ do} \)

\( \text{mentions} := \epsilon; \)

\( \text{for } e \leftarrow \text{extractors} \text{ do} \)

\( \text{if } \text{priority}(e) \text{ matches } i \text{ then} \)

\( \text{ms} := \text{extract}(e, \text{doc}, \text{state}); \)

\( \text{append(mentions, ms);} \)

\( \text{end} \)

\( \text{end} \)

\( \text{if } \text{isempty(mentions)} \text{ then} \)

\( \text{if } i \geq \text{miniterations} \text{ then} \)

\( \text{results} := \text{getmentions}(\text{state}); \)

\( \text{return globalAction(results, state);} \)

\( \text{end} \)

\( \text{else} \)

\( \text{update(state, mentions);} \)

\( \text{end} \)

\( \text{end} \)

**Algorithm 2:** Formal implementation of the Extractor Engine, which manages all of the token and dependency extractors used in the cascade. The value \textit{miniterations} holds the minimum number of iterations that need to be executed to ensure that every extractor gets executed at least once.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Done</td>
<td>The pattern matched successfully.</td>
</tr>
<tr>
<td>Pass</td>
<td>No-op.</td>
</tr>
<tr>
<td>Split(instruction1, instruction2)</td>
<td>Spawns two threads, one for each instruction. The thread for instruction1 has a higher priority than the one for instruction2.</td>
</tr>
<tr>
<td>SaveStart(name)</td>
<td>Starts capturing a token interval.</td>
</tr>
<tr>
<td>SaveEnd(name)</td>
<td>Stops capturing a token interval.</td>
</tr>
<tr>
<td>MatchToken(token-constraint)</td>
<td>Matches a token using a token constraint.</td>
</tr>
<tr>
<td>MatchMention(str-matcher, opt-name)</td>
<td>Matches and (optionally) captures a mention present in the state.</td>
</tr>
<tr>
<td>MatchSentenceStart</td>
<td>Asserts that the next token to explore is the first one in the sentence.</td>
</tr>
<tr>
<td>MatchSentenceEnd</td>
<td>Asserts that the last explored token is the last one in the sentence.</td>
</tr>
<tr>
<td>MatchLookAhead(pattern, negative)</td>
<td>Asserts that pattern matches what immediately follows the current token. Or that it doesn’t match, depending of the value of negative.</td>
</tr>
<tr>
<td>MatchLookBehind(pattern, negative)</td>
<td>Asserts that pattern matches what immediately precedes the current token. Or that it doesn’t match, depending of the value of negative.</td>
</tr>
</tbody>
</table>

Table 3.1: Virtual machine instructions.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutgoingWildcard</td>
<td>Follows any outgoing edge.</td>
</tr>
<tr>
<td>IncomingWildcard</td>
<td>Follows any incoming edge.</td>
</tr>
<tr>
<td>OutgoingDP(str-matcher)</td>
<td>Follows any outgoing edge whose label matches <em>str-matcher</em>.</td>
</tr>
<tr>
<td>IncomingDP(str-matcher)</td>
<td>Follows any incoming edge whose label matches <em>str-matcher</em>.</td>
</tr>
<tr>
<td>ConcatDP(lhs, rhs)</td>
<td>Evaluates <em>lhs</em>, uses its output as the input of <em>rhs</em>, and returns <em>rhs</em> results.</td>
</tr>
<tr>
<td>DisjunctiveDP(lhs, rhs)</td>
<td>Returns the union of evaluating <em>lhs</em> and <em>rhs</em>.</td>
</tr>
<tr>
<td>TokenConstraintDP(tok-constraint)</td>
<td>Filters all the tokens that are currently in the pattern’s state according to <em>tok-constraint</em>.</td>
</tr>
<tr>
<td>LookaroundDP(node, negative)</td>
<td>Evaluates the AST with root <em>node</em> in the current position. If it matches and <em>negative</em> is false, or it doesn’t match and <em>negative</em> is true, then the lookaround succeeds and matching of the whole pattern continues at the current position. Note that it does not distinguishes between lookahead and lookbehind, due to the nature of graph traversal.</td>
</tr>
<tr>
<td>OptionalDP(node)</td>
<td>Evaluates the AST with root <em>node</em> and returns its output concatenated with the current positions.</td>
</tr>
<tr>
<td>KleeneDP(node)</td>
<td>Evaluates <em>node</em> in a loop collecting its results, and returns the concatenation of the results at every iteration.</td>
</tr>
</tbody>
</table>

Table 3.3: Dependency pattern instructions
CHAPTER 4

APPLICATION TO BIOINFORMATICS

In the previous chapters we have described the design and implementation of ODIN, a domain independent IE framework. In this chapter we will describe how ODIN was used to build REACH, an IE system specifically tailored for the biomedical domain. REACH is a robust IE system that detects and extracts mentions of molecular interactions in the biomedical literature. It includes ODIN grammars for the recognition of biological entities (e.g., genes, proteins, protein families, simple chemicals), events over entities (e.g., biochemical reactions), and nested events that take other events as arguments (e.g., catalysis). Importantly, each of these grammars is compact, consisting of approximately 10 rules per category. This guarantees that the overall model is interpretable and can be easily modified and extended by domain experts.

Following is a description of the problem motivation, system implementation, and evaluation.

4.1 Motivation

The biomedical and life sciences literature is growing at a tremendous rate. PubMed\(^1\), a free indexing and search engine of biomedical and life sciences literature maintained by the United States National Library of Medicine, reflects this. Vardakas et al. (2015) point out that 7,364,633 journal articles were added to PubMed in the period 2004–2013. Figure 4.1 highlights this growth by plotting the number of indexed publications per year for the past 40 years.

This growth in the biomedical and life sciences literature increases the pressure for specialization. This specialization carries the risk of monopoly, monotony and isolation (Casadevall and Fang, 2014). For example, it is common that cancer

\(^1\)http://www.ncbi.nlm.nih.gov/pubmed
researchers focus their efforts on understanding a single cancer signaling pathway, or even a fragment of it, which impacts the design and development of many cancer drugs. As Hanahan and Weinberg (2000, 2011) explain in their seminal papers, most cancer drugs developed to date are directed toward specific molecular targets and signaling pathways that are involved in enabling particular biological capabilities. However, each of these capabilities are regulated by partially redundant signaling pathways, which causes the clinical responses to these drugs to be transitory, often followed by relapses.

To mitigate this type of reductionist science, the Defense Advanced Research Projects Agency (DARPA) created the Big Mechanism program (Cohen, 2015) to develop a holistic view of these cancer pathways through automated, large-scale reading and pathway assembly that both extends and amends already-existing human-curated models of protein interaction pathways. Two major requirements for accomplishing this ambitious goal are effective automatic reading of scientific publications, i.e., extracting biochemical interactions, and merging them into existing cancer models.

As part of this larger effort we developed REACH (REading and Assembling Contextual and Holistic mechanisms from text), an information extraction (IE) system designed to extract signaling pathway fragments from biomedical publications.

Figure 4.1: PubMed indexed publications per year. Note that these publication counts are not cumulative.
4.2 REACH

REACH is an IE system adapted for the biomedical domain. It uses a combination of rule-based and statistical techniques to read the content of a paper and produce mentions of molecular events that describe fragments of a signaling pathway. These mentions are constructed internally in a representation inspired by BioPAX, which is a language to represent biological pathways at the molecular and cellular level (Demir et al., 2010). Notably, REACH can capture mentions of interactions involving molecules and complexes, which are called Conversions in BioPAX, and are called simple events in REACH because their participants are always physical entities. REACH also captures interactions where one participant regulates another. These are called Controls in BioPAX and we refer to them as nested events because they can have other events as their participants. An important difference between REACH and BioPAX is that REACH supports mentions similar to BioPAX’s Control involving only physical entities. These interactions are not that useful when considered directly, but they provide valuable information about the interpretation of the mechanism that should be useful for assembling a complete pathway.

Additionally, REACH captures complex natural language phenomena. First, REACH contains a coreference resolution component that resolves anaphoric references for both entities (e.g., understanding that “the protein” resolves to “ASPP2”), and events (e.g., “this phosphorylation” corresponds to a phosphorylation reaction of a specific protein). Second, REACH models various forms of speculative statements (e.g., “It is thus tempting to speculate that beta catenin directly interacts with ZO-1”), and even more subtle phenomena such as reversing of nested controls polarity (e.g., “reduced Src activation results in decreased phosphorylation of ZO-1, VE-cadherin and beta-catenin”).
4.2.1 Preprocessing

In Chapter 2.2 we explained that ODIN can be used to extract information from text that has been preprocessed using common NLP tools. Because of this, the first step taken by REACH is the preprocessing of the text using NLP tools specifically modified for the biomedical domain. This NLP stack includes: sentence and word segmentation, POS tagging, and syntactic parsing.

The task of sentence and word segmentation consists of detecting both sentence and word boundaries in the input text. Notably, word segmentation of biomedical text is not handled correctly by open-domain tokenizers, mostly because of punctuation characters often used in the names of proteins and complexes. For this reason, a custom tokenizer was developed in-house, which conforms to the tokenization of the BioNLP corpus (Kim et al., 2009).

For POS tagging and syntactic parsing, REACH uses Stanford’s CoreNLP toolkit (Manning et al., 2014). It has been trained using a combination of two corpora: the Penn Treebank, a corpus of manually annotated documents of several genres like IBM computer manuals, nursing notes, Wall Street Journal articles, and transcribed telephone conversations (Marcus et al., 1993; Taylor et al., 2003); and the GENIA corpus, which is a manually annotated corpus of 2000 MEDLINE abstracts (Kim et al., 2003). Including the GENIA annotated documents as part of the parser’s training corpus makes the parser more robust to syntactic structures often found in biomedical literature.

The result of this preprocessing step is a document annotated with the syntactic structure of its content. We again point the reader to Figure 2.2 for an example of the kind of annotations obtained by this preprocessing step.

4.2.2 Entity Extraction

Next, a custom NER component is used to recognize mentions of relevant physical entities, i.e., species, cell lines, organs, cell types, families, cellular components, simple chemicals, sites, bioprocesses, and gene or gene products (this last category
This NER component works as follows:

1. A rule-based NER detects mentions of all the supported physical entities. This rule-based NER was generated automatically from the knowledge-bases (KBs) shown in Table 4.1. These KBs provide synonyms for each of the entities, which increases the NER coverage.

2. A CRF is used as a fallback to detect mentions that may have been missed by the rule-based NER. This CRF was trained on the BioCreAtIvE corpus (Hirschman et al., 2005) and only supports mentions of gene or gene products.

The result of these NER components is a sequence of tags in BIO notation, like those shown in Figure 2.2. REACH then proceeds to apply a collection of manually-built rules that perform three tasks:

1. Promote the entities encoded in BIO notation to ODIN mentions, using rules similar to the \texttt{ner} rule shown in Listing 2.1.

2. Applies a collection of manually-built rules designed to detect sites.

3. Applies rules that group entities when they are related according to some syntactic cues (e.g., a protein and its corresponding site). This grouping of related entities simplifies the event extraction process later on.

Lastly, REACH proceeds to ground the found physical entities. Grounding refers to the task of linking each entity to an entry in a database. Table 4.1 shows the supported entity types and their corresponding databases. Note that some entities can’t be grounded, either because it is expressed as an unsupported synonym or because REACH doesn’t support the corresponding database. In this situation, a unique identifier is generated and assigned to all entity mentions that correspond to this unknown entity.
<table>
<thead>
<tr>
<th>Entity Type</th>
<th>Database</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protein</td>
<td>UniProt Knowledgebase</td>
<td><a href="http://www.uniprot.org/">http://www.uniprot.org/</a></td>
</tr>
<tr>
<td>Protein families</td>
<td>InterPro</td>
<td><a href="http://www.ebi.ac.uk/interpro/">http://www.ebi.ac.uk/interpro/</a></td>
</tr>
<tr>
<td>Simple Chemicals</td>
<td>HMDB</td>
<td><a href="http://www.hmdb.ca/">http://www.hmdb.ca/</a></td>
</tr>
<tr>
<td>Simple Chemicals</td>
<td>ChEBI</td>
<td><a href="http://www.ebi.ac.uk/chebi/">http://www.ebi.ac.uk/chebi/</a></td>
</tr>
<tr>
<td>Sites</td>
<td>InterPro</td>
<td><a href="http://www.ebi.ac.uk/interpro/">http://www.ebi.ac.uk/interpro/</a></td>
</tr>
<tr>
<td>Subcellular locations</td>
<td>Gene Ontology</td>
<td><a href="http://amigo.geneontology.org/">http://amigo.geneontology.org/</a></td>
</tr>
<tr>
<td>Subcellular locations</td>
<td>UniProt subcellular locations</td>
<td><a href="http://www.uniprot.org/locations/">http://www.uniprot.org/locations/</a></td>
</tr>
<tr>
<td>Bioprocesses</td>
<td>Gene Ontology</td>
<td><a href="http://amigo.geneontology.org/">http://amigo.geneontology.org/</a></td>
</tr>
<tr>
<td>Bioprocesses</td>
<td>MeSH</td>
<td><a href="https://id.nlm.nih.gov/mesh/">https://id.nlm.nih.gov/mesh/</a></td>
</tr>
</tbody>
</table>

Note: Cell lines, cell types, and organs are not grounded to any database and get assigned a unique identifier with the “UAZ” namespace instead. The terms for cell lines and cell types were obtained from LINNAEUS (Gerner et al., 2010), and organs were obtained from http://www.rightdiagnosis.com/lists/organs.htm.

Table 4.1: Publicly available databases used to ground entities in the REACH system.

Next, REACH detects mentions of mutations and post-translational modifications (PTMs), and attaches them to the corresponding entities. This is done by applying another collection of manually-built ODIN rules.

4.2.3 Simple Events

Once REACH has determined which entities are mentioned in the text, it proceeds to extract the molecular interactions among them. These interactions are analogous to BioPAX’s Conversion. We refer to these interactions as simple events, alluding to the fact that their participants are physical entities. This is in contrast to nested events, which can take other events as their participants. Here we focus on simple events only, and nested events will be described below.

It is of note that few rules are needed to capture all simple events. This is achieved by first identifying several syntactic variations shared among the mentions of these events. We take advantage of this fact by parameterizing our rules and instantiating them using parameters like the possible triggers and compatible entity
types. Table 4.2 describes these syntactic variations and shows examples of each of them.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarative</td>
<td>The theme (the thing acted on by the verb) is the direct object of a verb.</td>
<td>“Smurf1 and Smurf2 degrade and ubiquitinate RhoA.”</td>
</tr>
<tr>
<td>Passive</td>
<td>The theme is the syntactic subject of a verb phrase.</td>
<td>“RhoA is ubiquitinated and degraded by Smurf1 and Smurf2.”</td>
</tr>
<tr>
<td>Prepositional Nominalization</td>
<td>The trigger is in noun form and entities are in prepositional phrases.</td>
<td>“The ubiquitination and degradation of RhoA by Smurf1 and Smurf2 increased.”</td>
</tr>
<tr>
<td>Object Nominalization</td>
<td>The trigger is in noun form and with the theme forms a noun-noun compound.</td>
<td>“RhoA ubiquitination and degradation by Smurf1 and Smurf2 increased.”</td>
</tr>
<tr>
<td>Subject Nominalization</td>
<td>The trigger is in noun form and with the cause forms a noun-noun compound.</td>
<td>“Smurf1 ubiquitination and degradation of RhoA increased.”</td>
</tr>
<tr>
<td>Subject Relative Clause</td>
<td>The trigger and theme are located in a relative clause which modifies the cause.</td>
<td>“Its many abnormal phenotypes can be rescued via Pde2, which specifically hydrolyzes cAMP.”</td>
</tr>
<tr>
<td>Object Relative Clause</td>
<td>The trigger and cause are located in a relative clause which modifies the theme.</td>
<td>“We measured transcription activation in the presence of cAMP, which is hydrolyzed by CRP.”</td>
</tr>
<tr>
<td>Subject Apposition</td>
<td>The cause is in an appositive phrase.</td>
<td>“Via yeast two-hybrid screening, we found that a novel protein, A20, binds to ABIN.”</td>
</tr>
<tr>
<td>Object Apposition</td>
<td>The theme is in an appositive phrase.</td>
<td>“Via yeast two-hybrid screening, we found that A20 binds to a novel protein, ABIN”</td>
</tr>
<tr>
<td>Paraphrastic Causative</td>
<td>The trigger is separated from an entity by a verb.</td>
<td>“Smurf1 causes the degradation of RhoA.”</td>
</tr>
</tbody>
</table>

Combinations of the above syntactic variations are also considered. For example, an appositive subject relative plus passivization: “Pde2, which has been found to hydrolyze Ras, activates MEK.”

Table 4.2: Common syntactic variations shared among events.

Twelve different types of simple events are supported. Nine of them are biochemical reactions: phosphorylation, ubiquitination, hydroxylation, sumoylation,
glycosylation, acetylation, farnesylation, ribosylation, and methylation. All of these reactions involve the covalent modification of a protein. The difference between these reactions and the PTMs mentioned in the previous step is that these reactions refer to the act of modifying the protein by attaching a functional group to it, and the PTMs described in the previous step refer to proteins that have already been modified. The three remaining simple events are: translocation, which refers to the act of transporting an entity from one cell location to another; binding, which is the process of assembling a complex from two or more proteins; and hydrolysis, which is the cleavage of chemical bonds by the addition of water.

REACH uses ODIN’s rules to detect and extract mentions of these molecular interactions. It does so by matching phrases that cue the mention of one of these interactions in the text. We refer to these phrases as triggers. Once a trigger is matched, the syntactic patterns that form part of the rule are used to retrieve the participating entities. These entities are then assigned a named role according to their involvement in the interaction.

Listing 4.1 shows an example of a parameterized syntax-based rule that is designed for simple events. Listing 4.2 shows an example of a parameterized surface-based rule, also for capturing simple events.

4.2.4 Nested Events

Once the extraction of simple events has concluded, the extraction of nested events begins. REACH recognizes positive and negative regulations, which map directly to BioPAX Controls. They represent explicitly the promotion or inhibition of a reaction. This means that the “controlled” participant must be either a simple or a nested event.

REACH also recognizes positive and negative activations, which are structurally very similar to regulations with the exception that the “controlled” participant is a physical entity. Activations are not representable in BioPAX, but REACH supports them because they are abstractions frequently used to summarize the result of a sequence of steps in a signaling pathway. These activations are not as useful as reg-
ulations when considered in isolation, but they provide valuable information about the author's interpretation of the mechanism. This information should be useful when attempting to assemble a complete pathway from the extracted fragments.

Since the participants of an activation are physical entities, they could be considered simple events. However, we consider them part of the nested events because of their similarity to regulations, which allows us to reuse the same parameterized rules by just specifying different type constraints on their participants (e.g., event vs physical entity).

Nested events also conform to the syntactic patterns shown in Table 4.2. An example of a parameterized rule for nested events is shown in Listing 4.3. Note that the triggers and argument types are expressed as variables. This allows us to use the

### Listing 4.1: Example simple-event parameterized syntax rule

```plaintext
- name: ${eventName}_syntax_1_noun
  priority: ${priority}
  label: ${label}
  action: ${action}
  pattern: |
    # match a nominal trigger that does not start with "de"
    # e.g., we want to match "phosphorylation"
    # but not "dephosphorylation"
    # also, should not be preceded by a protein mention
    # (with optional site)
    trigger =
      (?<! @Gene_or_gene_product @Site?)
      [lemma=/${nominalTriggerLemma}/ & !word=/^de/]
    # match the theme of the event
    # e.g., the target of the phosphorylation
    theme: BioChemicalEntity =
      prep_of appos? /nn| conj_ (and|or|nor)|cc/{,2}
    # (optionally) match the cause of the event
    # note that this is actually the controller of a regulation
    # but sometimes it is easier to match everything together
    cause: BioChemicalEntity? =
      (<dobj)? (prep_by|agent)|nn|prep_of prep_by
      /nn| conj_ (and|or|nor)|cc/{,2}
      |
      poss
    # (optionally) match the event site
    site: Site? = (/ prep_ / nn{,2}){1,2}
```

```
same template to instantiate positive regulations with controlled events, or positive activations with controlled entities.

4.2.5 Complex Natural Language Phenomena

In addition of the event and entity extraction grammars described previously, REACH also recognizes complex phenomena that are hard to express with ODIN rules. The phenomena captured by REACH are coreference resolution, negation, hedging, and reverse polarity caused by nested controls.

REACH implements a sieve-based coreference resolution component designed specifically for the biomedical domain. This component recognizes entity and event coreference both in the same sentence or across sentences. This component’s architecture is beyond the scope of this Chapter, but we point to (Bell et al., 2016) for a detailed explanation.

Negation, hedging, and reverse polarity will be explained here since they all rely on similar machinery and are a good illustration of things that can’t be currently expressed declaratively by ODIN, and may inform the development of future ODIN extensions.
Negations occur frequently in biological literature, since it is used by authors to explicitly state that a reaction does not occur (e.g., “ZAP70 does not induce higher levels of TRIM tyrosine phosphorylation”). Negated mentions are marked as such by REACH.

Hedging is another common linguistic phenomenon that is used by authors to reduce the strength of their statements. Generally, people expect scientific language to be factual. In reality, however, authors usually soften their claims, either to avoid appearing overconfident or to reduce the risk of opposition. Authors usually use hedges to refer to speculative statements, or to distinguish known facts from their own claims (e.g., “we hypothesize . . .”, “these results suggest . . .”). REACH recognizes when an interaction mention is part of a speculative statement and it keeps track of which mentions are speculative.

A more subtle phenomenon consists of statements that involve nested controls with different polarities (e.g., “decreased PTPN13 expression enhances EphrinB1 and Erk1 phosphorylation”). REACH can handle this phenomena and interprets
the true meaning of these statements.

All these types of phenomena are detected by traversing the path that connects the trigger and the arguments in the dependency graph, keeping track of words that are considered negations and hedges. Additionally, adjectival modifiers that connect to the path at any point are also considered, in order to reverse polarities. Figure 4.2 shows the dependency parse of a sentence that includes this latter phenomena. In this example the word “enhances” seems to indicate that PTPN13 up-regulates the phosphorylation of both EphrinB1 and Erk1, but if we follow the amod dependency we find that it is the “decrease” of PTPN13 that enhances the phosphorylations. This is interpreted by REACH as a polarity flip, as shown in Figure 4.3, where the verb “enhances” is instead interpreted in a negative regulation relationship.

![Figure 4.2: Sentence with nested controls](image)

![Figure 4.3: Control with reversed polarity.](image)

4.3 Evaluation

REACH was developed and evaluated within the Big Mechanism program. In this section, we describe the most recent large-scale evaluation of IE performers in the program, in which participants had to extract mechanistic information from a thousand papers about the Ras signaling pathway over the course of a week. The evaluation took place in the summer of 2015. Four consortia, each one potentially containing multiple teams, participated in the evaluation.
In this evaluation, systems had to output pathway fragments encoded into “index cards”, where each index card encodes a single biochemical reaction with its controller (if available). Listing 4.4 shows an example of an index card where participant_a encodes the controller (ZAP70 in this case), participant_b captures the protein, gene, or chemical entity being operated on (TRIM), interaction_type indicates the type of reaction captured by the index card (adds_modification), and the modifications field contains the modifications involved in this card (phosphorylation). The negative_information field captures the fact that the text evidence indicates that the reaction does not occur.

```json
{
  "pmc_id": "PMC2212462",
  "evidence": [
    "ZAP70 does not induce higher levels of TRIM tyrosine phosphorylation"
  ],
  "extracted_information": {
    "interaction_type": "adds_modification",
    "modifications": [
      {
        "modification_type": "phosphorylation"
      }
    ],
    "negative_information": true,
    "participant_a": {
      "entity_text": "ZAP70",
      "entity_type": "protein",
      "identifier": "Uniprot:P43403"
    },
    "participant_b": {
      "entity_text": "TRIM",
      "entity_type": "protein",
      "identifier": "Uniprot:Q6PIZ9"
    }
  }
}
```

Listing 4.4: An example of an index card describing a phosphorylation reaction. Some index card fields were omitted for brevity.

This IE evaluation focused on the correctness of the information extracted, and also on how well the information fits a preexisting model centered on the Ras protein.
Because the essence of this work is information extraction, we focus on the former part of the evaluation in this section.

The participating consortia followed different approaches for this IE evaluation. For anonymity, we do not identify the participating consortia by name, but briefly describe their approaches. Team 1 implemented a pipeline of machine learning components that addressed various aspects of the task, such as identifying interaction types, interaction participants, etc. Teams 2 and 3 implemented a hybrid approach, where they used machine learning to construct a semantic representation of the text (Allen et al., 2008; Banarescu et al., 2012), and a rule-based component to extract domain-specific information from this open-domain semantic representation. Team 4 included two systems: REACH and a rule-based system (Team 4(B)).

Information extraction systems are typically evaluated using precision (i.e., percentage of correct index cards from the index cards produced by the system) and recall (i.e., percentage of correct index cards extracted out of all the index cards in the paper collection). Given that this evaluation was a large-scale experiment (over a thousand papers), recall could not be computed directly. Instead, in this DARPA evaluation, recall was approximated using the system throughput. Throughput, and several precision scores, were estimated using the following algorithm:

1. Cards that do not conform to the index card format were eliminated.

2. Then, cards that do not encode complete biochemical phenomena or interactions supported by experiments in the corresponding paper were skipped. The following criteria were considered for skipping index cards:

   - Activations without participant_a (e.g., “Ras was increased”).
   - Translocations without destination (e.g., “Ras was transported from the membrane”).
   - Cards that addressed interactions not covered in the experimental results of the corresponding paper (e.g., interactions extracted from discussions of previous work).
3. Similar findings were deduplicated. For example, two index cards describing the same interaction were counted once.

4. A sample from the remaining cards was manually scored by an expert panel on the accuracy of their extracted information (e.g., is the interaction type correct?, are the participants correct?). The scores were normalized to be between zero and one by dividing by the total number of information bits expected per card.

5. *Throughput* was computed as total number of validated cards produced (step 1) multiplied by the proportion of cards with positive scores in the evaluated sample (step 4) divided by the number of evaluation days. Informally, throughput is the estimated number of useful cards produced per day.

6. A first lenient precision measure, called “generous” precision, was defined as the number of cards with a positive score divided by the number of scored cards (ignoring skipped and repeated cards). In other words, generous precision is the proportion of index cards that were considered useful by the expert panel.

7. A second precision measure, called “strict” precision, was implemented as the sum of the scores divided by the number of scored cards. Intuitively, strict precision is the proportion of cards that were completely correct.

The results for all the participating teams are shown in Table 4.3. The table shows that Team 4, which includes REACH, produced approximately 20 thousand validated cards (after step 1 of the scoring process). This led to the second highest throughput in the evaluation; Team 2 had the highest throughput at the cost of a much lower precision. REACH was responsible for 80% of Team 4’s throughput. In other words, even without considering the ensemble system of Team 4, REACH had the second highest throughput of the evaluation.

REACH had the highest strict precision of all the participating systems, 2% higher than Team 3, and the second highest generous precision, approximately 1% below Team 3. It is worth noting that REACH had a throughput approximately
<table>
<thead>
<tr>
<th>Team</th>
<th>Valid Cards</th>
<th>Throughput</th>
<th>Strict Precision</th>
<th>Generous Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team 1</td>
<td>5,936</td>
<td>110</td>
<td>54.69%</td>
<td>62.50%</td>
</tr>
<tr>
<td>Team 2</td>
<td>37,887</td>
<td>975</td>
<td>35.24%</td>
<td>42.07%</td>
</tr>
<tr>
<td>Team 3</td>
<td>8,522</td>
<td>242</td>
<td>56.76%</td>
<td>75.68%</td>
</tr>
<tr>
<td>Team 4</td>
<td>20,924</td>
<td>944</td>
<td>50.36%</td>
<td>66.42%</td>
</tr>
<tr>
<td>REACH</td>
<td>15,361</td>
<td>760</td>
<td>58.76%</td>
<td>74.23%</td>
</tr>
<tr>
<td>Team 4(B)</td>
<td>5,563</td>
<td>189</td>
<td>30.00%</td>
<td>47.50%</td>
</tr>
</tbody>
</table>

Table notes: The table shows the performance of the four teams, including throughput, strict precision, and generous precision. Team 4 was an ensemble system consisting of REACH and Team 4(B).

Table 4.3: Results of the summer 2015 Big Mechanism evaluation.

four times higher than Team 3. We believe this is a positive result considering the compactness of our IE model: our grammars combined contain 154 rules, of which 122 are event rules. It is also worth noting that 78% of the event rules are syntactic patterns, which shows the importance of syntax in this work.

Figure 4.4 summarizes the performance of all teams according to their throughput and generous precision. Note that only Team 4 and REACH are located in the top-right quadrant, which highlights that they have the best combination of precision and throughput. Also of note is the fact that by combining the results of REACH and Team 4(B), the throughput of Team 4 continues to improve, at a small cost of precision loss.

While these scores are not perfect (strict precision is still below 60%), these numbers compare favorably with other IE evaluations that looked at the entire content of papers (Kim et al., 2013). The best performing system in this evaluation used a supervised machine learning approach, and had a precision of 58% and a recall of 45%. These results are not directly comparable because the datasets are different, but they suggest that REACH compares well against supervised learning approaches. Importantly, DARPA implemented another evaluation exercise, where human domain experts were asked to annotate the same data. Human and machine performance were compared taking three different error types into account:
identification of participant A, interaction type, and grounding. The results show that REACH complements human curators. Particularly, humans were more accurate than REACH at identifying participant A (4% vs 35% error rate), humans and REACH accuracy at identifying the interaction type were comparable (7% vs 5% error rate), and REACH was much better at grounding entities (68% vs 18% error rate) (Korves et al., 2016).

Figure 4.4: Summary of the Big Mechanism evaluation. Team 4 was an ensemble system consisting of REACH and Team 4(B).

Table 4.4 shows the strict precision achieved by REACH per index card type. Note that some index cards “flatten” several REACH event types. In particular, adds_modification and inhibits_modification collapse a positive/negative regulation with its controlled reaction, e.g., phosphorylation. The other index card types map directly to REACH event types. increases_activity and decreases_activity correspond to positive/negative activations. binds maps to a complex assembly event, and translocates captures a protein translocation event. The table indicates that REACH obtains the lowest performance on
increases_activity index cards. A post-hoc error analysis indicated that the majority of these errors were caused by incorrect handling of hedging phenomena such as multiple negations.

<table>
<thead>
<tr>
<th>Card Type</th>
<th>Strict Precision</th>
<th>Num. Cards in Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>adds_modification</td>
<td>66.41%</td>
<td>32</td>
</tr>
<tr>
<td>inhibits_modification</td>
<td>50.00%</td>
<td>1</td>
</tr>
<tr>
<td>increases_activity</td>
<td>47.73%</td>
<td>33</td>
</tr>
<tr>
<td>decreases_activity</td>
<td>71.43%</td>
<td>7</td>
</tr>
<tr>
<td>binds</td>
<td>59.52%</td>
<td>21</td>
</tr>
<tr>
<td>translocates</td>
<td>100.00%</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.4: Strict precision for specific event types as determined during the Big Mechanism 2015 evaluation.

Table 4.3 lists REACH’s throughput at 760 index cards per day. This number is misleading: this was computed simply by dividing by the total number of evaluation days (7). In reality, REACH is much faster. On average, on a machine with Intel Xeon CPU (2.4GHz) with 48 cores, REACH processes a complete PubMed paper in 4.5 seconds (this includes all the necessary preprocessing such as syntactic analysis).

The whole system was implemented using 154 unique rule templates that a user would interact with, some of which are instantiated internally multiple times. As described in Chapter 2.3.5, templates have variables that can be specified to instantiate the automata. For example, the 26 templates that define simple events (as shown in Table 4.5) are used to generate the 234 automata used for the extraction of all nine biochemical conversions (i.e., phosphorylation, ubiquitination, hydroxylation, sumoylation, glycosylation, acetylation, farnesylation, ribosylation, and methylation) by specifying the appropriate values for the template variables. The number of templates and their types are shown in Table 4.5. In Figure 4.6 we show the distribution of the contribution of each rule to the generated output. Note that the distribution appears to be approximately Zipfian.

The rule Positive_activation_syntax_1_verb (Listing 4.5) is the one with the highest number of matches. It captures Positive_activation events by matching trigger words such as “activates” or “promotes”, and then finds the controlled and
controlled arguments by following the specified syntactic paths. An example of 
a match for this rule is “Endothelin-1 activates eNOS”, where Endothelin-1 is the 
controller argument and eNOS is the controlled argument.

The rule Acetylation_token_4_noun (Listing 4.6) is one of the rules with the 
least number of matches. It is a surface rule that matches a sequence of tokens 
starting with the trigger “acetylation”, followed by some optional tokens and then it 
captures a site and the theme of the acetylation event. An example of a match for 
this rule is “acetylation at this specific lysine of cdk2”, where cdk2 is the theme and 
lysine is the site.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Surface</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entities</td>
<td>0</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Generic entities</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Modifications</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Mutants</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td><strong>Total entities</strong></td>
<td>0</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Simple events</td>
<td>15</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>Binding</td>
<td>30</td>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>Hydrolysis</td>
<td>8</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Translocation</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Positive regulation</td>
<td>16</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>Negative regulation</td>
<td>14</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td><strong>Total events</strong></td>
<td>95</td>
<td>27</td>
<td>122</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>95</td>
<td>59</td>
<td>154</td>
</tr>
</tbody>
</table>

Figure 4.5: Number of rule templates in REACH’s grammar
Figure 4.6: Distribution of the number of entity and event matches per instantiated rule.

Listing 4.5: Rule with the most number of matches.

```plaintext
- name: Positive_activation_syntax_1_verb
  priority: 8
  label: Positive_activation
  action: mkActivation
  pattern: |
    trigger =
      [word=/\(?i\)\^\(acceler|activat|aid|allow|augment|direct|elev|
elicit|enabl|enhanc|increas|induc|initi|modul|necess|
overexpress|potenti|produc|prolong|promot|rais|reactivat|
rescu|respons|restor|re-express|retr|sequest|signal|
stimul|support|synerg|synthes|trigger|up-regul|upregul)/ &
tag=/\^V\|RB/]
      [lemma=/\^\(activ|regul\)/ & tag=/\^V/])?

controlled: BioEntity =
  prepc_by?
  (dobj | xcomp | ccomp)
  /conj|dep|dobj|cc|nn|prep_of|prep_in|amod/{,2}
  (>> [word=by])

controller: PossibleController =
  <xcomp?
    (nsubj | agent | <vmod)
    /appos|nn|conj_|cc|prep_of|prep_in/{,2}
```

Listing 4.5: Rule with the most number of matches.
Listing 4.6: Rule with the least number of matches.
CHAPTER 5

LEARNING INTERPRETABLE MODELS

In previous chapters we have described the design and implementation of the ODIN IE framework. We have also described the design and implementation of the REACH IE high-throughput system for the biomedical domain. REACH uses manually-built ODIN rules to extract molecular interactions from biomedical literature. We showed in Chapter 4 that REACH requires few rules to support all the molecular interactions of interest by relying on shared syntactic structures. All the rules, however, were manually built.

In this chapter we will describe a method to extract rules from a statistical model that was trained on an annotated corpus. This enables us to combine the advantages of machine learning with the interpretability of our models, allowing users to further edit the IE models to fix or improve them.

5.1 The BioNLP’09 Shared Task

As we explained in Chapter 4, the amount of biomedical publications is exploding (see Figure 4.1 for a visualization of the publication rate through time). Consequently, it is of interest to the biomedical research community to automate the reading of the literature in order to extract the relevant information and make it available in a structured fashion. The BioNLP’09 Shared Task was organized to advance research in this area (Kim et al., 2009).

The main concern of BioNLP’09 is the extraction of molecular interactions from the biomedical literature. These interactions are referred to as events or bio-events in the context of this shared task. The shared task was divided in three tasks, of which only the first one was mandatory for all participants. The first (required) task, called “core event extraction” consists of identifying interactions between pro-
## Type Primary arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Primary arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene_expression</td>
<td>Theme(Protein)</td>
</tr>
<tr>
<td>Transcription</td>
<td>Theme(Protein)</td>
</tr>
<tr>
<td>Protein_catabolism</td>
<td>Theme(Protein)</td>
</tr>
<tr>
<td>Phosphorylation</td>
<td>Theme(Protein)</td>
</tr>
<tr>
<td>Localization</td>
<td>Theme(Protein)</td>
</tr>
<tr>
<td>Binding</td>
<td>Theme(Protein)+</td>
</tr>
<tr>
<td>Regulation</td>
<td>Theme(Protein/Event), Cause(Protein/Event)</td>
</tr>
<tr>
<td>Positive_regulation</td>
<td>Theme(Protein/Event), Cause(Protein/Event)</td>
</tr>
<tr>
<td>Negative_regulation</td>
<td>Theme(Protein/Event), Cause(Protein/Event)</td>
</tr>
</tbody>
</table>

Figure 5.1: BioNLP’09 event types. Secondary arguments are omitted since they are not involved in the “core event extraction” task.

teins/genes. The gold named entities corresponding to proteins/genes (without distinction) were provided by the task organizers. The second (optional) task, called “event enrichment” involves the augmentation of the events found in the required task by identifying named entities that were not provided (e.g., sites) and assigning them to the corresponding events. The third (optional) task is the “negation and speculation recognition”, which, as its name implies, is concerned with the detection of negations and speculations regarding the events extracted during the required task. We will focus on the “core event extraction” task only.

The BioNLP’09 dataset consists of 1,210 paper abstracts divided in training (800 abstracts), development (150 abstracts), and test (260 abstracts) subsets. The dataset includes a set of gold protein annotations to be used by the participants, allowing them to focus on the event extraction task. Nine events were addressed, all shown in Table 5.1.

### 5.2 The statistical model

We implemented a supervised IE system inspired by the one proposed by Björne et al. (2009). The system’s architecture is shown in Figure 5.2. It consists of two multiclass logistic regression classifiers, where the first classifier detects and labels trigger words in the input text, and the second one labels tuples of the form (trigger,
CD2 signaling induces phosphorylation of CREB in primary lymphocytes.

Figure 5.2: Architecture of the statistical model for the BioNLP 2009 core event extraction task.

We used logistic regression classifiers for this experiment, but any feature-based classifier that assigns weights to features could be used instead (e.g., perceptron, linear SVM).

5.2.1 Trigger extraction

The first multiclass logistic regression classifies labels each word in the text as being a trigger of some specific event class (event classes are shown in Table 5.1) or they get assigned a negative label when they are not detected as being triggers. Differing from the work the system presented by Björne et al. (2009), we don’t use combined classes for words that may be triggers for several different event classes, choosing the most likely one instead.

Our trigger classifier considers different kinds of features to detect a trigger and predict its label.

**Surface features** These features include the original and lemmatized words, and
the presence of the word in a gazetteer of known triggers. These features are generated for the word being classified, as well as the words surrounding it inside a window of $n$ tokens. Also, bag-of-words features are generated for the window and for the sentence as a whole.

**Syntactic features** The syntactic chains connected to the token are used as features. Two different versions are generated, the chain with the dependencies only, and also the chains with the word at every hop.

**Entity features** The number of entities surrounding the token, both inside a window and in the sentence as a whole.

5.2.2 Relation extraction

The relation extraction is performed by pairing all detected triggers either with the named entities or with other triggers. These pairs are then classified into one of the possible relations or a negative label indicating that there is no relation between the pair.

The features are used for classifying tuples are:

**Syntactic features** These features are based on the shortest path connecting the two mentions in the dependency graph. Several versions of the shortest path are used: a lexicalized one, an unlexicalized one, and path fragments of different sizes.

**Surface features** These features include: the order of the mentions, their distance in terms of tokens, the number of entities and triggers in the sentence, the parts of speech and words of the mentions, and the number of triggers and entities between the mentions.

**Consistency features** Including the labels of the mentions, and the labels of their superclasses. Useful for representing the semantic restrictions of event arguments.
**Graph features** The parent, children, and siblings of the mentions, according to the dependency graph.

5.2.3 Feature to rule translation limitations

These features are common to most machine learning IE systems. However, not all of them can be represented as rules with the current implementation of our rule language. For example, we cannot completely handle bag-of-word features, or syntactic features in surface patterns. Additionally, features like dependency n-grams that form part of the shortest path between entities cannot be represented, since our syntactic patterns need to be anchored at both ends. Finally, any feature relying on counting occurrences of tokens or entities cannot be handled by our framework. These limitations affect the conversion of features for classifying triggers and relations. Features that cannot be handled by our framework were not included in the statistical system, which degraded its performance as compared with state-of-the-art systems. Potential extensions to the rule language are discussed in Chapter 7.2.3.

5.3 Converting the statistical model to rules

Both classifiers were trained using $L_1$ regularization in order to reduce the number of features (Ng, 2004). Once the classifiers had been trained, we proceeded to convert the remaining features into ODIN rules and converting the feature weights into votes.

5.3.1 Converting features to rules

In general, the features previously introduced consist of conjunctions of information bits, each of which corresponds to a different rule fragment. For example, for the classification of event participants, one such conjunction captures the type of the expected trigger (e.g., Phosphorylation), combined with the syntactic path from the trigger to the participant candidate (e.g., an outgoing passive nominal subject – `nsubjpass`), and a semantic constraint on the type of named entity of the partici-
pant (e.g., Protein). These are immediately translatable to rules, as illustrated in Figure 5.3.

Importantly, the rules encode output information as well, e.g., the recognized event participant serves as a theme for a Phosphorylation event in Figure 5.3. At this stage, this information is exhaustively generated from all possible classifier labels (e.g., for the classification of event participants these labels are the cartesian product of \{theme, cause\} and possible event labels \{Phosphorylation, Binding, ...\}). Of course, some of these outputs do not apply. For example, it is highly unlikely that the rule shown in Figure 5.3 produces the cause of a Regulation event. We quantify the confidence in these outputs in the next stage of the algorithm.

Figure 5.3: Example of rule built from the conjunction of the trigger “phosphorylation”, the outgoing nsubjpass, and the semantic argument constraint of Protein.

5.3.2 Converting weights to votes

Feature weights are unbounded continuous values that are difficult to interpret and modify. They are, however, useful for resolving conflicts. For example, if the system learned that a word could be a trigger for two different events, by assigning a higher weight to one event class over the others it would represent the system’s higher confidence on that particular prediction, before considering other evidence.

Therefore, the next step in the conversion of the statistical model to rules is the conversion of weights into a more interpretable representation in which each rule has been assigned a (positive or negative) number of votes. When rules find a match, they cast their votes, which can then be used to select the most likely of several
conflicting matches. Votes are discrete values, similar to a Likert scale (Likert, 1932). This is achieved by binning the weight values into discrete categories. Several methods have been proposed for selecting the number of bins in a histogram, for example (Sturges, 1926; Doane, 1976; Freedman and Diaconis, 1981). We use the rule proposed by Scott (1979):

$$ h = 3.5\hat{\sigma}n^{-1/3} $$

(5.1)

where $h$ is the estimated bin width, $n$ is the sample size, and $\hat{\sigma}$ is the estimated standard deviation. We use Scott’s rule since it gives a good compromise between retaining most of the information in the weights while minimizing the number of bins. The binned weights for trigger and relation features are shown in Figures 5.4 and 5.5 respectively.

![Trigger features](image)

Figure 5.4: Discretization of the trigger weights learned by the logistic regression classifier.

5.4 Model editing

In this approach, we emphasize interpretability of the learned rules, asserting that human users should be able to read and understand the rules so as to be able to make
improvements. To evaluate the potential of this, the generated rules were given to two expert linguists who were allotted one hour to improve them. They received the rules only, without the training data. Their suggested changes to the rules can be grouped into three categories: 1) Generalize syntactic patterns by adding optional traversals, 2) Make rules robust to common parsing mistakes, and 3) Improve overall readability. Representative examples of these changes can be seen in Table 5.1, along with motivation for each change.

5.5 Evaluation

The results of the IE system using logistic regression with $L_2$ regularization are shown in Table 5.6. This system uses 1,190,029 distinct features.
<table>
<thead>
<tr>
<th>Event Class</th>
<th>Recall</th>
<th>Precision</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene_expression</td>
<td>57.58</td>
<td>74.28</td>
<td>64.87</td>
</tr>
<tr>
<td>Transcription</td>
<td>40.24</td>
<td>57.89</td>
<td>47.48</td>
</tr>
<tr>
<td>Protein_catabolism</td>
<td>61.90</td>
<td>86.67</td>
<td>72.22</td>
</tr>
<tr>
<td>Phosphorylation</td>
<td>51.06</td>
<td>82.76</td>
<td>63.16</td>
</tr>
<tr>
<td>Localization</td>
<td>47.17</td>
<td>92.59</td>
<td>62.50</td>
</tr>
<tr>
<td>Binding</td>
<td>18.15</td>
<td>34.62</td>
<td>23.81</td>
</tr>
<tr>
<td>Event Total</td>
<td>42.75</td>
<td>64.61</td>
<td>51.45</td>
</tr>
<tr>
<td>Regulation</td>
<td>8.28</td>
<td>40.00</td>
<td>13.73</td>
</tr>
<tr>
<td>Positive_regulation</td>
<td>17.18</td>
<td>42.74</td>
<td>24.51</td>
</tr>
<tr>
<td>Negative_regulation</td>
<td>7.14</td>
<td>40.00</td>
<td>12.12</td>
</tr>
<tr>
<td>Regulation Total</td>
<td>13.65</td>
<td>42.14</td>
<td>20.62</td>
</tr>
<tr>
<td>All Total</td>
<td>26.77</td>
<td>56.22</td>
<td>36.27</td>
</tr>
</tbody>
</table>

Figure 5.6: Performance of the IE pipeline comprised of logistic regression with $L_2$ regularization across all event types.

We then trained the same system using $L_1$ regularization as a form of feature selection, reducing the number of features from 1,190,029 to 10,926. The system’s performance is shown in Table 5.7. This drastic reduction of features came with the cost of a slightly reduced F1 score.

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Recall</th>
<th>Precision</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene_expression</td>
<td>58.71</td>
<td>78.28</td>
<td>67.09</td>
</tr>
<tr>
<td>Transcription</td>
<td>37.80</td>
<td>55.36</td>
<td>44.93</td>
</tr>
<tr>
<td>Protein_catabolism</td>
<td>61.90</td>
<td>86.67</td>
<td>72.22</td>
</tr>
<tr>
<td>Phosphorylation</td>
<td>46.81</td>
<td>84.62</td>
<td>60.27</td>
</tr>
<tr>
<td>Localization</td>
<td>56.60</td>
<td>88.24</td>
<td>68.97</td>
</tr>
<tr>
<td>Binding</td>
<td>16.13</td>
<td>33.33</td>
<td>21.74</td>
</tr>
<tr>
<td>Event Total</td>
<td>42.75</td>
<td>66.60</td>
<td>52.08</td>
</tr>
<tr>
<td>Regulation</td>
<td>8.88</td>
<td>65.22</td>
<td>15.62</td>
</tr>
<tr>
<td>Positive_regulation</td>
<td>13.13</td>
<td>40.50</td>
<td>19.83</td>
</tr>
<tr>
<td>Negative_regulation</td>
<td>8.16</td>
<td>55.17</td>
<td>14.22</td>
</tr>
<tr>
<td>Regulation Total</td>
<td>11.41</td>
<td>44.44</td>
<td>18.15</td>
</tr>
<tr>
<td>All Total</td>
<td>25.54</td>
<td>59.35</td>
<td>35.72</td>
</tr>
</tbody>
</table>

Figure 5.7: Performance of the IE pipeline comprised of logistic regression with $L_1$ regularization across all event types.

The results of the rule-based system which uses rules and votes extracted from
the logistic regressions trained with $L_1$ regularization are shown in Table 5.8. With this system, F1 score decreased further. Note that we could achieve the same F1 score as the system in Table 5.7 by decreasing bin size, but this would drastically decrease interpretability. Recall that we try to find a balance between fidelity to the original weights and a reduced number of bins using Scott’s rule. Potentially, better methods could be explored to improve these results.

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Recall</th>
<th>Precision</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene_expression</td>
<td>55.34</td>
<td>76.95</td>
<td>64.38</td>
</tr>
<tr>
<td>Transcription</td>
<td>28.05</td>
<td>53.49</td>
<td>36.80</td>
</tr>
<tr>
<td>Protein_catabolism</td>
<td>57.14</td>
<td>85.71</td>
<td>68.57</td>
</tr>
<tr>
<td>Phosphorylation</td>
<td>40.43</td>
<td>90.48</td>
<td>55.88</td>
</tr>
<tr>
<td>Localization</td>
<td>45.28</td>
<td>88.89</td>
<td>60.00</td>
</tr>
<tr>
<td>Binding</td>
<td>12.90</td>
<td>33.33</td>
<td>18.60</td>
</tr>
<tr>
<td>Event Total</td>
<td>38.04</td>
<td>67.18</td>
<td>48.58</td>
</tr>
<tr>
<td>Regulation</td>
<td>5.33</td>
<td>75.00</td>
<td>9.94</td>
</tr>
<tr>
<td>Positive_regulation</td>
<td>10.70</td>
<td>48.89</td>
<td>17.55</td>
</tr>
<tr>
<td>Negative_regulation</td>
<td>5.61</td>
<td>55.00</td>
<td>10.19</td>
</tr>
<tr>
<td>Regulation Total</td>
<td>8.76</td>
<td>51.50</td>
<td>14.97</td>
</tr>
<tr>
<td>All Total</td>
<td>21.97</td>
<td>62.98</td>
<td>32.57</td>
</tr>
</tbody>
</table>

Figure 5.8: Performance of the IE pipeline comprised of automatically extracted rules from the logistic regression classifiers across all event types.

Finally, the results of the system after the rules were edited following the expert’s suggestions are shown in Table 5.9. Several of the suggestions involved removing or collapsing rules, which decreased the number of rules from 10,926 to 8,868. Here we see that the interpretability of the learned grammar allowed for users to make changes to the rules which both improved readability and performance. Comparing the F1 score of this edited system against that of the system with $L_1$ regularization, we see that much of the lost performance has been regained while further reducing the size of the grammar.
<table>
<thead>
<tr>
<th>Event Class</th>
<th>Recall</th>
<th>Precision</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene_expression</td>
<td>60.39</td>
<td>70.49</td>
<td>65.05</td>
</tr>
<tr>
<td>Transcription</td>
<td>31.71</td>
<td>57.78</td>
<td>40.94</td>
</tr>
<tr>
<td>Protein_catabolism</td>
<td>61.90</td>
<td>81.25</td>
<td>70.27</td>
</tr>
<tr>
<td>Phosphorylation</td>
<td>42.55</td>
<td>86.96</td>
<td>57.14</td>
</tr>
<tr>
<td>Localization</td>
<td>45.28</td>
<td>88.89</td>
<td>60.00</td>
</tr>
<tr>
<td>Binding</td>
<td>22.18</td>
<td>23.50</td>
<td>22.82</td>
</tr>
<tr>
<td>Event Total</td>
<td>43.74</td>
<td>54.31</td>
<td>48.46</td>
</tr>
<tr>
<td>Regulation</td>
<td>10.06</td>
<td>40.48</td>
<td>16.11</td>
</tr>
<tr>
<td>Positive_regulation</td>
<td>12.80</td>
<td>44.89</td>
<td>19.92</td>
</tr>
<tr>
<td>Negative_regulation</td>
<td>10.71</td>
<td>51.22</td>
<td>17.72</td>
</tr>
<tr>
<td>Regulation Total</td>
<td>11.91</td>
<td>45.17</td>
<td>18.86</td>
</tr>
<tr>
<td>All Total</td>
<td>26.27</td>
<td>51.71</td>
<td>34.84</td>
</tr>
</tbody>
</table>

Figure 5.9: Performance of the IE pipeline comprised of automatically extracted rules from the logistic regression classifiers, after human intervention.

Figure 5.10 shows a learning curve for the logistic regression and rule-based system. This curve seems to be asymptoting, which may indicate that the system is limited by the features that it uses. By extending the rule language, as discussed in Chapter 7.2.3, we expect that the increase in the number of abstracts used in training would yield further gains.
Figure 5.10: Learning curve showing the change in F1 performance as a function of the amount of training data. We compare the performance of both the logistic regression with $L_1$ regularization (shown in red) and the final rule-learning system prior to the expert modifications (shown in blue).
<table>
<thead>
<tr>
<th>Suggested Change</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generalization</strong></td>
<td></td>
</tr>
<tr>
<td>Add /conj_((and</td>
<td>or</td>
</tr>
<tr>
<td>Ensure that all syntactic paths end in appos?</td>
<td>Handle optional apposition to increase rule coverage.</td>
</tr>
<tr>
<td>Replace all specific named entities with their label.</td>
<td>For example, in rules like [word=phosphorylates] (?=MEK) that reference a specific protein, replace the protein with the label Protein. This improves rule generalization and reduces the total number of rules.</td>
</tr>
<tr>
<td>Make the &gt;nn optional in Theme:Protein = &gt;nsubjpass &gt;nn</td>
<td>If a syntactic path ends in an nn dependency, it can be made optional to increase coverage, and reduce the set of rules.</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td></td>
</tr>
<tr>
<td>Replace agent with /(^{agent</td>
<td>prep_by}$$/</td>
</tr>
<tr>
<td>Change ccomp to /(^{c</td>
<td>x}comp/ and acomp to /(^{a</td>
</tr>
<tr>
<td><strong>Readability</strong></td>
<td></td>
</tr>
<tr>
<td>Merge rules when possible, e.g. prep_of, prep_of nn, prep_of appos become prep_of (nn? appos</td>
<td>nn appos? nn?)?</td>
</tr>
<tr>
<td>Eliminate trigger rules that are not sufficiently discriminative (e.g., ?&lt;=([lemma=&quot;be&quot;]) [tag=/(^{V</td>
<td>N</td>
</tr>
<tr>
<td>Only use lemma and tag features in trigger rules for simple events (other than transcription and binding).</td>
<td>This simplification retains sufficiently discriminative constraints only.</td>
</tr>
<tr>
<td>Remove redundant constraints</td>
<td>For example, in patterns like [incoming=nsubj] &amp; tag=/(^{N}/ the POS tag is redundant because it is implicitly defined through the incoming dependency (nominal subject).</td>
</tr>
<tr>
<td>Group trigger rules in priorities according to their likelihood of signalling a given event, and add a constraint that says the trigger cannot overlap with existing triggers</td>
<td>Some trigger words may be ambiguous (i.e., they may signal different event types) and by organizing the rules in priorities we can give preference to some of them over others.</td>
</tr>
</tbody>
</table>

Table 5.1: Representative examples of the rule changes suggested by linguistic experts.
CHAPTER 6

RELATED WORK

As we mentioned in Chapter 1.2, information extraction is the task of scanning unstructured text, finding the knowledge contained in it, and returning a structured representation of it. This structured representation of the knowledge contained in the raw text can then be indexed or manipulated for further processing.

In this chapter we will describe the main techniques currently in use for information extraction.

6.1 Information extraction with machine learning

As noted by Chiticariu et al. (2013), most of the academic research in the topic focuses on machine learning approaches for IE. These approaches vary depending on the amount of available training data that they require. In this section we will discuss the supervised approach (useful when we have lots of annotated data available), semi-supervised methods (for situations in which we have limited examples available), and unsupervised techniques (when we don’t even know the relations we are looking for). However, the body of work in machine learning for information extraction is very large, and therefore we invite the interested reader to also consult a dedicated survey such as the one by Turmo et al. (2006).

6.1.1 Supervised

Supervised information extraction was the first widely investigated natural language processing task. It started with the Message Understanding Conference (MUC) (Grishman and Sundheim, 1996), which ran for seven years and focused on event extraction, ranging from extraction of terrorism activities to airplane crashes. MUC-1
was organized in 1987 and had neither a specified output format nor a formal evaluation. The subsequent MUCs were organized as a template-filling task, where each template had a predetermined number of slots to be filled automatically by the participating systems from the provided texts. The evaluation was also formalized, in order to better comparing the participants using the precision and recall metrics.

After MUC ended in 1997, in 1999 another shared task was developed which continued to promote the development of information extraction technologies. Automatic Content Extraction (ACE) was organized by the National Institute of Standards and Technology (NIST) with the stated purpose of:

>[developing] technology to automatically infer from human language data the entities being mentioned, the relations among these entities that are directly expressed, and the events in which these entities participate.

(Doddington et al., 2004)

These tasks promoted the advancement of information extraction by defining tasks and evaluation methods that allowed participants to be compared fairly. Machine learning approaches were possible, since the tasks provided hand-made annotations that the participants could use to develop their systems.

The traditional approach in fully supervised systems is to use a large collection of hand-annotated texts to train a classifier. This classifier can then be used to automatically annotate unseen documents (Jurafsky and Martin, 2008), that is, to label the relevant entities and the relations between them.

This approach usually involves a classifier pipeline in which a first classifier can be used to detect named-entities, and then a second classifier can be used to detect if there is a valid relation between two entities, and if so, label the relation.

An example of one such fully supervised IE system for the biomedical domain is described by Björne et al. (2009), which was the basis for the pipeline detailed in Chapter 5.2. Björne et al.’s system was built for the BioNLP’09 shared task (Kim et al., 2009) in which the named-entities (i.e., proteins) are provided as part of the system’s input in addition to the raw text, removing the need for named-entity
recognition. Their system works as follows:

1. Preprocess input text (e.g., sentence splitting, parsing).

2. Detect triggers using a multi-class SVM.

3. Detect relations of the form trigger-entity for simple events or trigger-trigger for nested events using a multi-class SVM.

4. Rule-based semantic post-processing (e.g., detect invalid argument combinations).

The system described in Chapter 5.2 follows the same steps although it differs on the implementation details.

6.1.2 Semi Supervised

As long as sufficient annotated data is available, fully supervised systems are easily trainable and achieve good performance. This annotated data, however, is expensive to generate and so typically fully supervised systems do not achieve their full potential due to lack of training data. Another category of system, referred to as semi-supervised, overcomes this problem by finding indirect sources of training data. These systems can either use a small number of manually generated examples (bootstrapping), or use external knowledge sources to generate noisy annotations that can be used to train a supervised classifier (distant supervision). Each of these approaches are described in the sections that follow.

**Bootstrapping**

Bootstrapping refers to an IE approach that doesn’t require a corpus of labeled examples to train a classifier. Instead, it uses a few seed examples of the relations of interest, which are used as a guide to find more relations. As an illustrative example, if you have a labeled tuple representing book authorship such as *(Jules Verne, Around the World in 80 Days)*, you can then extract a set of sentences which
contain the tuple elements. If the sentence “Jules Verne wrote *Around the World in 80 Days*” is retrieved, the system would recognize a pattern (AUTHOR wrote BOOK), which can be used to retrieve more tuples, which are then used to generate more patterns, and so on.

This approach was employed by Brin (1998), who introduced DIPRE (Dual Iterative Pattern Relation Expansion) as a method for bootstrapping relation extraction patterns from a small set of tuples (seeds). Algorithm 3 details this approach with pseudocode.

| Input: seeds: user-provided seed tuples |
| Input: C: corpus |
| Output: $R'$: extracted tuples |

\[
R' \leftarrow \text{seeds};
\]

**repeat**

\[
O \leftarrow \text{FindOccurrences}(R', C);
\]

\[
P \leftarrow \text{GeneratePatterns}(O);
\]

\[
R' \leftarrow \text{Search}(C, P);
\]

**until** $R'$ is large enough;

**Algorithm 3:** Dual Iterative Pattern Relation Expansion, a bootstrapping algorithm for extracting relations.

Snowball (Agichtein and Gravano, 2000) builds on DIPRE by estimating the confidence of a pattern

\[
Conf(P) = \frac{P_{\text{pos}}}{P_{\text{pos}} + P_{\text{neg}}} \quad (6.1)
\]

where $P_{\text{pos}}$ is the number of positive matches for $P$ and $P_{\text{neg}}$ is the number of negative matches. It then estimates the confidence of an extracted tuple by combining the confidence of the patterns that matched it

\[
Conf(T) = 1 - \prod (1 - Conf(P_i)) \quad (6.2)
\]
Distant supervision

Distant supervision for IE, introduced by Craven et al. (1999), combines the advantages of bootstrapping and supervised learning. It accomplishes this by using an external database which contains examples from the domain of interest, which takes the place of the seeds in bootstrapping. Since databases usually contain many records, and each of these records is used as a starting seed, there is no need to bootstrap new tuples to increase coverage. The database is aligned with free text to generate a collection of noisy examples, which are then used to train a supervised classifier.

Mintz et al. (2009) proposed a system that begins by using Freebase (Bollacker et al., 2008) to get relation examples. Then, the input free text is scanned using an NER system to find all entity mentions. If a sentence contains entities corresponding to one of the example relations, then that sentence is considered a positive example of that relation. The features extracted from all sentences corresponding to a specific Freebase relation are combined into a single feature vector. Once a feature vector has been constructed for each of the relations of interest, they are passed to a logistic regression classifier. This classifier can then be used to extract the relations of interest from previously unseen text.

One problem to be aware of is the fact that the generated examples are usually noisy, since they are generated automatically. For example, if we see the fact (Barack Obama, president-of, USA) in our database, we may label the sentence “Barack Obama was born in the USA” as a positive example of the president-of relation, which is clearly wrong. Therefore, there is interest in developing methods that are resilient to noisy data, like the one proposed by Surdeanu et al. (2012).

The Knowledge Base Population shared task (KBP) (Surdeanu, 2013), and particularly the Slot Filling track, is the standard task for distant supervision for information extraction.
6.1.3 Unsupervised

For some tasks, however, there is no labeled data and not even a list of desired relations. For example, we may want to extract relations from the web, which is an immense collection of heterogeneous documents that contains an unknown and potentially massive number of relations. Open IE was introduced by Banko et al. (2007) to tackle this task. The relations found by Open IE are encoded as strings, which then have to be mapped to some ontology in order to be used (e.g., indexed in a database).

One Open IE system is ReVerb (Fader et al., 2011), which from a text such as “Hudson was born in Hampstead, which is a suburb of London” extracts the tuples (Hudson, was born in, Hampstead) and (Hampstead, is a suburb of, London). It works by applying the following steps:

- Identify relation phrases
- Select noun phrases for relation arguments
- Assign confidence score to the relation using a logistic regression classifier

Systems such as these are able to discover new relations, but the relations are not grounded, so the same relation expressed in different natural language could produce different tuples.

6.2 Rule Based IE

The use of patterns, or rules, for IE was first proposed by Hearst (1992), who proposed a set of lexico-syntactic patterns to detect hyponymy relations. These patterns are:

- NP such as \{NP ,\} \{ and | or \} NP
- such NP as \{NP ,\} \{ and | or \} NP
- NP \{, NP \} \{,\} and other NP
• NP {, NP} {,} or other NP

• NP {,} including {NP ,}* {and | or} NP

• NP {,} especially {NP ,}* {and | or} NP

FASTUS (Appelt et al., 1993) is a rule-based IE system implemented as a cascade of finite state automata (FSA), which has become a common architecture for subsequent rule-based IE systems. This approach has proven capable of producing fast and robust parsers for unstructured text (Abney, 1996). The success of FSA cascades continues even today with systems such as GATE (Cunningham et al., 2002).

FASTUS introduced the Common Pattern Specification Language (CPSL) as a formalism for specifying cascaded FSA grammars (Appelt and Onyshkevych, 1998). A grammar in CPSL is specified by defining a cascade of finite state transducers that work by matching regular expressions over the lexical features of the input symbols.

Other languages that follow CPSL’s approach of matching regular expressions over the lexical features of the input are GATE’s Java Annotation Patterns Engine (JAPE) (Thakker et al., 2009), Stanford’s TokensRegex (Chang and Manning, 2014), and the Allen Institute for Artificial Intelligence taggers\(^1\). ODIN follows in this lineage; however, unlike these approaches, ODIN allows the mixing of both surface-and syntax-based rules in the same grammar. Furthermore, because ODIN builds on top of simple and proven syntactic dependency representations (de Marneffe and Manning, 2008), the learning curve for ODIN is short.

SProUT’s XTDL (Piskorski et al., 2004) extends CPSL’s approach using unification-based grammars to give the language more expressivity. However, this introduces additional complexity in the language. In our opinion, this is not always necessary in domain-specific scenarios, where lexical information fully disambiguates the context. Furthermore, similar to most previous work, XTDL does not support syntactic patterns.

\(^1\) https://github.com/allenai/taggers
Rule: University1

(Token.string == "University")
(Token.string == "of")
{Lookup.minorType == city}
):orgName
-->
:orgName.Organisation =
{kind = "university", rule = "University1"}  

Figure 6.1: Example JAPE rule. From https://gate.ac.uk/sale/talks/gate-course-may10/track-1/module-3-jape/module-3-jape.pdf  

pp :> morph & [ POS Prep,
    SURFACE #prep,
    INFL [ CASE #c ] ]
    (morph & [ POS Det,
        INFL [ CASE #c,
            NUMBER #n,
            GENDER #g ] ) ?
    (morph & [ POS Adjective,
        INFL [ CASE #c,
            NUMBER #n,
            GENDER #g ] ] ) *
    (morph & [ POS Noun,
        SURFACE #noun,
        INFL [ CASE #c,
            NUMBER #n,
            GENDER #g ] ] )

-> phrase & [ CAT pp,
    PREP #prep,
    AGR agr & [ CASE #c,
        NUMBER #n,
        GENDER #g]
    CORE_NP #core_np],
    where #core_np=Append(#det," ",#noun).  

Figure 6.2: Example XTDL rule. From http://sprout.dfki.de/XTDLSampleRule.html
create view Person as
select S.name as name
from (  
  ( select CombineSpans(F.name, C.name) as name  
   from First F, Caps C  
   where FollowsTok(F.name, C.name, 0, 0))
union all
  ( select CombineSpans(F.name, L.name) as name  
   from First F, Last L  
   where FollowsTok(F.name, L.name, 0, 0))
union all
  ( select *  
   from First F )
) S
consolidate on name;

Figure 6.3: Example AQL rule. From http://www.emnlp2015.org/tutorials/15/15_OptionalAttachment.pdf

From the languages that support syntax, Stanford’s Tregex matches patterns over constituency trees (Levy and Andrew, 2006). For ODIN we chose to use dependency-based syntax for two reasons: simplicity of representation, and availability of linear-time parsers (Chen and Manning, 2014). Semgrex is a language that modifies Tregex to operate over dependency graphs (Chambers et al., 2007)\(^2\). However, neither of these languages support cascaded FSA.

In a departure from CPSL, IBM’s SystemT is a rule-based IE system that uses the AQL language, which is inspired from SQL (Li et al., 2011). AQL is a powerful language that implements an IE algebra (Reiss et al., 2008). However, in our opinion, this loses some of the simplicity that ODIN enjoys.

\(^2\)See also Semgrex’s online documentation: http://nlp.stanford.edu/software/tregex.shtml
6.3 Conclusion

To summarize, this chapter highlighted that an immense body of work in NLP has focused on information extraction. However, despite these impressive efforts, no rule-based IE framework that addresses all the requirements of real-world information extraction exists. This motivated our proposed approach, which uses a declarative rule-based language to express extraction patterns over multiple representations of the input text (Chapter 2), and an efficient runtime system that compiles and executes the rules expressed in the language (Chapter 3).

The work presented in this dissertation builds upon all this previous work in two ways: first, we propose a declarative rule-based approach that is inspired by all the body of work starting with FASTUS and ending with GATE; second, our approach could be combined with any of the ML approaches to learn the relevant rules.
7.1 Summary

As we explained in the Introduction, the vast amount of unstructured text available requires the construction of information extraction (IE) systems in order to generate structured representations of the knowledge contained in this text. Particularly, we are interested in developing interpretable models that can handle this task so that humans can understand the models and continue to improve them over time.

This work contributed a rule-based IE framework composed of a declarative language which can operate over both surface and syntactic representations of the text as well as an efficient runtime system capable of compile and executing the models expressed in our language. We used this IE framework to build a state-of-the-art system designed to read biomedical papers and extract cell-signalling pathways, handling complex natural language phenomena like coreference and hedging. We also propose a rule-learning procedure that combines the advantages of machine learning with the interpretability of our rule-based framework and show that it is able to generate models that humans can later improve. While these are themselves valuable contributions, we describe some possible future work that could improve the interpretability of the language, the performance of the runtime system, and the utility of the outputs.

7.2 Future Work

The future work in this line of research centers on three main areas: adding to the existing capabilities of the framework, improving the rule-learning procedure, and making use of the system output in a downstream task.
7.2.1 Assembly of knowledge graph

In this work we have been concerned with the extraction of knowledge fragments, such as “A phosphorylates B”, from unstructured text. The next step is to build a knowledge graph that implements a causal model from the individual fragments. We already have some assembly capabilities through ODIN’s support for nested structures, and we can also assemble fragments across sentences using coreference resolution. However, we would like to read a collection of documents and construct a single graph that summarizes the knowledge contained in the whole collection.

Particularly, causal relations can be used to assemble a graph representing the causal interactions between different concepts or entities. This could be done by using specific lexical and syntactic features of causal statements. Such a graph would be useful for several applications including question answering systems and information retrieval systems which have more semantically complex queries.

In the case of scientific publications, we could incorporate knowledge about how the papers cite each other (Zhu et al., 2015; Valenzuela et al., 2015).

Another issue in automatic assembly is the automatic ontology acquisition. The system that we describe in Chapter 4 targets a specific domain with a previously specified ontology for the entities and events involved. However, when developing systems for underspecified domains, we would like to infer the ontology automatically in order to be able to correctly assemble the extracted fragments into a knowledge graph.

7.2.2 Rule learning

Recall that the rule learning system described in Chapter 5 is optimized for a balance between performance and interpretability. However, through extensions including distant supervision, rule merging, and extensions to the ODIN framework we expect that we can boost both.
Distant Supervision

We showed in Chapter 5 that we can use statistical models to learn rules from annotated examples. Since the system is fully supervised, however, we are currently limited by the amount of available training data. Through the use of distant supervision, we could either increase the size of the available annotated data or generate a new dataset when no annotations are available. This generated data is usually noisy, but there are methods for handling this noise such as those found in Surdeanu et al. (2012). In addition to the expected performance gain from simply having more examples for training the classifier, with more data we could potentially discover additional lexico-syntactic patterns that signal the occurrence of the events of interest, further increasing our system’s IE coverage.

In the biomedical domain there are several existing knowledge bases (e.g., PathwayCommons (Cerami et al., 2011), Reactome (Matthews et al., 2009)) which could be used as input for distant supervision, along with large sources of unannotated text (e.g., PMC open access subset\(^1\)). Distant supervision over these resources could potentially further improve the REACH system’s (described in Chapter 4) performance.

Rule Merging

One of the main goals of the rule learning system is generating interpretable rules. This is necessary so that human users can understand the generated grammars, allowing them to debug the grammars and improve them over time. One critical component of interpretability is the number of rules that compose a single grammar. If there are too many, a human user will not be able to sift through them all. While we have reduced the number of rules generated by our system through aggressive feature selection (going from 1.2M rules to approximately 10k, dropping two orders of magnitude), there are still far too many rules for a user to realistically read and understand. Generated rules can be collapsed in several ways in order to reduce the

\(^1\)http://www.ncbi.nlm.nih.gov/pmc/tools/openftlist/
total number of rules in a grammar, greatly improving the interpretability.

First, rules with compatible triggers and arguments can be merged into a single rule. For instance if there are two separate rules using the same trigger and each expresses complementary arguments, then they could potentially be merged into a single rule. To illustrate, consider the sentence, “A phosphorylates B,” where A and B are the cause and theme of the phosphorylation event, respectively. Our current system would learn one rule for each of the arguments, but potentially a single rule which contains both could capture the same information. This not only reduces the number of rules, but the collapsed rule is more precise because it captures the fact that either all the arguments need to be found or that some of the arguments can only occur in the presence of the others. (See Chapter 2 for examples of how rules can express optional arguments.)

Second, patterns that overlap can be merged into a single pattern by constructing the minimal finite-state automata (Daciuk et al., 2000), and then generating a regular expression for it. For example, if we have two distinct but overlapping syntactic patterns such as \texttt{>prep_of} and \texttt{>prep_of >nn}, we can merge them into a single pattern by making use of ODIN’s quantifiers (e.g., \texttt{>prep_of >nn?}). This merging should not be applied indiscriminantly, but instead some heuristic should be considered in order to not obfuscate the patterns and maintain readability.

Reducing the number of rules in these ways, would improve the systems interpretability by increasing the likelihood of a user being able to find problematic or insufficient rules and make the necessary modifications.

7.2.3 ODIN Extensions

Our IE framework can be extended both to increase its expressivity as well as to improve its performance.

Currently, we can express lexical and syntactic patterns in our declarative language. However, there are natural language phenomena that can not be captured. For example, we can’t currently match words by their semantic similarity. By adding new language capabilities, we can increase its expressivity in order to allow complex
patterns that are currently not supported. This would potentially also improve our rule-learning method because we can only currently use features that are able to be turned into rules in our language. By augmenting the expressivity of our language, we would be able to include additional features that are commonly used in information extraction systems but cannot currently be expressed in our language.

Aside from the rule-learning component, increasing the expressivity of the language would allow users to write novel patterns. For example, adding support for word embeddings (Mikolov et al., 2013) would potentially allow matching tokens by their semantic similarity. Other features that can be added are the support of numerical comparisons and the use of syntactic pattern assertions in surface patterns, such as requiring that a given token has a specific syntactic path attached to it.

Another possible extension to the framework would be to infer the best order for rule application automatically. Currently, if the rule priorities are not specified manually, we apply heuristics to execute the rules the required number of times to either produce an output or else to guarantee that an output cannot be found. We can instead implement an algorithm that infers the optimal rule execution order (e.g., Rete (Forgy, 1982)), removing the need for the user to specify the order manually (which can be error prone) as well as the potential inefficiency introduced by our heuristics.
This appendix contains the grammars that define ODIN’s extractor language. The grammars are written in Backus-Naur Form (BNF), with terminals enclosed in quotes, non-terminals enclosed in angle brackets, and optional items enclosed in square brackets. The * and + characters denote the Kleene star and Kleene plus respectively.

A.1 Token Constraint Grammar

The following grammar describes Odin’s token constraints. A token constraint is a boolean expression over a token’s lexical, morphological, or semantic attributes.

\[
\langle TokenConstraint \rangle ::= \' [ \langle DisjunctiveConstraint \rangle ] \' \\
\langle DisjunctiveConstraint \rangle ::= \\
\langle ConjunctiveConstraint \rangle (\' | \langle ConjunctiveConstraint \rangle )* \\
\langle ConjunctiveConstraint \rangle ::= \\
\langle NegatedConstraint \rangle (\' & \langle NegatedConstraint \rangle )* \\
\langle NegatedConstraint \rangle ::= \[ \' ! \] (AtomicConstraint) \\
\langle AtomicConstraint \rangle ::= \langle FieldConstraint \rangle | \' ( \langle DisjunctiveConstraint \rangle \' ) \\
\langle FieldConstraint \rangle ::= \langle FieldName \rangle \' = \' (StringMatcher) \\
\langle FieldName \rangle ::= \\
\' word \' \\
| \' lemma \' \\
| \' tag \'
| 'entity'  |
| 'chunk'   |
| 'incoming'|
| 'outgoing'|
| 'mention' |

\[ \langle \text{StringMatcher} \rangle ::= \langle \text{ExactStringMatcher} \rangle \mid \langle \text{RegexStringMatcher} \rangle \]

\[ \langle \text{ExactStringMatcher} \rangle ::= \langle \text{StringLiteral} \rangle \]

\[ \langle \text{StringLiteral} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{SingleQuoteString} \rangle \mid \langle \text{DoubleQuoteString} \rangle \]

\[ \langle \text{RegexStringMatcher} \rangle ::= \langle \text{RegexLiteral} \rangle \]

A.2 Token Pattern Grammar

In this section we describe the BNF grammar for ODIN’s token patterns, i.e., the surface rules. Token patterns support several advanced features like lazy and greedy quantifiers, named captures of both mentions and sequences of tokens with the ability to share names among the captures, and zero-width assertions.

\[ \langle \text{TokenPattern} \rangle ::= \langle \text{DisjunctiveTokenPattern} \rangle \]

\[ \langle \text{DisjunctiveTokenPattern} \rangle ::= \]
\[ \quad \langle \text{ConcatenatedTokenPattern} \rangle \ ('\mid' \langle \text{ConcatenatedTokenPattern} \rangle)^* \]

\[ \langle \text{ConcatenatedTokenPattern} \rangle ::= \]
\[ \quad \langle \text{QuantifiedTokenPattern} \rangle \langle \text{QuantifiedTokenPattern} \rangle^* \]

\[ \langle \text{QuantifiedTokenPattern} \rangle ::= \]
\[ \quad \langle \text{AtomicTokenPattern} \rangle \]
\[ \mid \langle \text{RepeatedTokenPattern} \rangle \]
\[ \mid \langle \text{RangeTokenPattern} \rangle \]

\[ \langle \text{AtomicTokenPattern} \rangle ::= \]
\[ \quad \langle \text{SingleTokenPattern} \rangle \]
\(\text{RepeatedTokenPattern} ::= \)
\(\text{AtomicTokenPattern} \; (\,??\; |\; ^{*}?\; |\; ^{*}?\; |\; ^{*}\; |\; ^{*})\)

\(\text{RangeTokenPattern} ::= \)
\(\text{AtomicTokenPattern} \; \{'\; \langle\text{number}\; ,\; \langle\text{number}\; \}\}'\)
\(\; |\; \text{AtomicTokenPattern} \; \{\; \langle\text{number}\; ,\; \}\}'\)
\(\; |\; \text{AtomicTokenPattern} \; \{\; \langle\text{number}\; \}\}'\)
\(\; |\; \text{AtomicTokenPattern} \; \{\; \langle\text{number}\; \}\}'\)

\(\text{SingleTokenPattern} ::= \text{StringMatcher} \; |\; \text{TokenConstraint}\)

\(\text{MentionTokenPattern} ::= \)
\('\@' \; [\text{StringLiteral} \; ':'\] \; \text{ExactStringMatcher}\)

\(\text{CaptureTokenPattern} ::= \)
\('(?<\; \langle\text{identifier}\; \}'\; \text{DisjunctiveTokenPattern} \; ')}\)

\(\text{AssertionTokenPattern} ::= \)
\(\langle\text{SentenceStartTokenAssertion}\rangle\)
\(\; |\; \langle\text{SentenceEndTokenAssertion}\rangle\)
\(\; |\; \langle\text{LookaheadTokenAssertion}\rangle\)
\(\; |\; \langle\text{LookbehindTokenAssertion}\rangle\)

\(\text{SentenceStartTokenAssertion} ::= \; '^''\)

\(\text{SentenceEndTokenAssertion} ::= \; '$''\)

\(\text{LookaheadTokenAssertion} ::= \)
\(\langle\text{SentenceStartTokenAssertion}\rangle \; |\; \langle\text{SentenceEndTokenAssertion}\rangle \; |\; \langle\text{LookaheadTokenAssertion}\rangle \; |\; \langle\text{LookbehindTokenAssertion}\rangle\)

\(\text{LookbehindTokenAssertion} ::= \)
\(\langle\text{SentenceStartTokenAssertion}\rangle \; |\; \langle\text{SentenceEndTokenAssertion}\rangle \; |\; \langle\text{LookaheadTokenAssertion}\rangle \; |\; \langle\text{LookbehindTokenAssertion}\rangle\)
A.3 Dependency Pattern Grammar

This BNF grammar describes the syntax for ODIN’s dependency patterns. These patterns are applied over a dependency graph. Notable features include the usual regex quantifiers (although there is no lazy/greedy distinction), lookaround assertions (again, no distinction between lookahead and lookbehind), and they can pack/unpack arguments using argument quantifiers as explained in Section 2.3.4 Token constraints are also supported as a way of adding lexical constraints at any step of the path.

\[
\langle \text{DependencyPattern} \rangle ::= \\
\quad \langle \text{TriggerPatternDependencyPattern} \rangle \\
\quad | \quad \langle \text{TriggerMentionDependencyPattern} \rangle
\]

\[
\langle \text{TriggerPatternDependencyPattern} \rangle ::= \\
\quad 'trigger' '=' \langle \text{TokenPattern} \rangle \langle \text{ArgPattern} \rangle^+
\]

\[
\langle \text{TriggerMentionDependencyPattern} \rangle ::= \\
\quad \langle \text{identifier} \rangle ':' \langle \text{identifier} \rangle \langle \text{ArgPattern} \rangle^+
\]

\[
\langle \text{ArgPattern} \rangle ::= \\
\quad \langle \text{identifier} \rangle ':' \langle \text{identifier} \rangle ['*' | '+' | '?' | '{' \langle \text{number} \rangle '}'] '=' \langle \text{DisjunctiveDependencyPattern} \rangle
\]

\[
\langle \text{DisjunctiveDependencyPattern} \rangle ::= \\
\quad \langle \text{ConcatenatedDependencyPattern} \rangle ('\mid' \langle \text{ConcatenatedDependencyPattern} \rangle)^*
\]

\[
\langle \text{ConcatenatedDependencyPattern} \rangle ::= \\
\quad \langle \text{StepDependencyPattern} \rangle \langle \text{StepDependencyPattern} \rangle^*
\]

\[
\langle \text{StepDependencyPattern} \rangle ::= \\
\quad \langle \text{FilterDependencyPattern} \rangle \\
\quad | \quad \langle \text{TraversalDependencyPattern} \rangle
\]

\[
\langle \text{FilterDependencyPattern} \rangle ::= \langle \text{TokenConstraint} \rangle
\]
\(\langle \text{TraversalDependencyPattern} \rangle ::=\)
\(\langle \text{AtomicDependencyPattern} \rangle\)
\(\mid \langle \text{RangeDependencyPattern} \rangle\)
\(\mid \langle \text{QuantifiedDependencyPattern} \rangle\)

\(\langle \text{QuantifiedDependencyPattern} \rangle ::= \langle \text{AtomicDependencyPattern} \rangle \ (‘?’ | ‘*’ | ‘+’)\)

\(\langle \text{RangeDependencyPattern} \rangle ::=\)
\(\langle \text{AtomicDependencyPattern} \rangle \ ‘{‘ }\langle \text{number} \ ‘,’ \langle \text{number} \ ‘}’\)
\(\mid \langle \text{AtomicDependencyPattern} \rangle \ ‘{‘ }\langle \text{number} \ ‘,’ \ ‘}’\)
\(\mid \langle \text{AtomicDependencyPattern} \rangle \ ‘{‘ }\langle \text{number} \ ‘,’ \ ‘}’\)
\(\mid \langle \text{AtomicDependencyPattern} \rangle \ ‘{‘ }\langle \text{number} \ ‘}’\)

\(\langle \text{LookaroundDependencyPattern} \rangle ::=\)
\(‘(?=’ | ‘(?!) \langle \text{DisjunctiveDependencyPattern} \rangle ‘)’\)

\(\langle \text{AtomicDependencyPattern} \rangle ::=\)
\(\langle \text{OutgoingDependencyPattern} \rangle\)
\(\mid \langle \text{IncomingDependencyPattern} \rangle\)
\(\mid \langle \text{LookaroundDependencyPattern} \rangle\)
\(\mid ‘(\ ‘\langle \text{DisjunctiveDependencyPattern} \rangle ‘)’\)

\(\langle \text{OutgoingDependencyPattern} \rangle ::= ‘>>’ | ‘[>’ \langle \text{StringMatcher} \rangle\)

\(\langle \text{IncomingDependencyPattern} \rangle ::= ‘<<’ | ‘<’ \langle \text{StringMatcher} \rangle\)
In Chapter 2 we described how to write token and dependency patterns using the ODIN information extraction framework. In this Appendix, we will go through the set up of a complete system using ODIN to extract marriage events from free text. In Listing B.1, we define a minimal grammar which we assume to be saved to the current working directory in a file named `marriage.yml`. 
Listing B.1: An example of a small set of rules designed to capture a marriage event and its participants. The rules that run in priority 1 make use of the output of an NER system to capture mentions for Person, Location, and Date. According to the rule “marriage-syntax-1”, a Marriage event requires at least one spouse and may optionally have a Date and Location.

We can now use our marriage.yml event grammar to extract mentions from free text. Listing B.2 shows a simple Scala program to do just this. We instanti-
ate a CoreNLPProcessor which uses Stanford’s CoreNLP to parse and annotate the provided text with the attributes required by ODIN (see Table 2.2 for a list of the relevant attributes). This annotated text is stored in a Document which is then passed to ODIN through the EventEngine.extractFrom() method. Finally we collect the Marriage mentions found by ODIN and display them using the Mention.json() method, which converts the mention into a JSON representation. A portion of this output is shown in Listing B.3.
import edu.arizona.sista.odin._
import edu.arizona.sista.processors.corenlp.CoreNLPProcessor

object SimpleExample extends App {
  // two example sentences
  val text = ""
    |John and Alice got married in Vegas last March.
    |Caesar and Cleopatra never married.
    |I think they got married.
    |Zarbon and Frederick will marry next summer.
    |She and Burt finally got married.
    |Simon and Samantha got married in Tucson on March 12, 2010 at the Desert Museum.
  "".stripMargin

  // read rules from general-rules.yml file in resources
  val source = io.Source.fromFile("marriage.yml")
  val rules = source.mkString
  source.close()

  // Create a simple engine without custom actions
  val extractor = ExtractorEngine(rules)

  // annotate the sentences
  val proc = new CoreNLPProcessor
  val doc = proc.annotate(text)

  // extract mentions from annotated document
  val mentions = extractor
    .extractFrom(doc)
    .filter(_.matches "Marriage")

  // display the mentions
  mentions.foreach{ m => println(m.json(pretty=true)) }
}

Listing B.2: A simple Scala program using the marriage.yml rules shown in Listing B.1. These rules do not call any custom actions. For an explanation of how to link rules to custom actions, please refer to Section ??.
Listing B.3: An example of one of the captured Marriage mentions outputted as JSON. The "characterOffsets" field corresponds to the original text provided in Listing B.2.
An example of a complete project including details on how to specify ODIN’s dependencies is available here:

https://github.com/clulab/odin-examples

Readers seeking a starting point for their own projects can refer to the code in the linked repository which contains working examples covering both simple and complex scenarios.
As described in Chapter 2, ODIN rules produce mentions, which store all the relevant information generated during the match. This is sufficient for most common usages of ODIN, but sometimes this information requires some changes. For example, one could use coreference resolution to replace event arguments that are pronouns with their nominal antecedents, as indicated by the coreference resolution component. This is not easily done though rules, and this is when actions become necessary.

Actions are Scala methods (implemented by the domain developer!) that can be applied by ODIN’s runtime engine to the resulting Mentions after matching the rule. An Action has the type signature shown in Listing C.1.

```scala
1 def action(mentions: Seq[Mention], state: State): Seq[Mention]
```

Listing C.1: Signature of action methods.

A rule will first try to apply its pattern to a sentence. Any matches will be sent to the corresponding action as a Mention sequence. If an action is not explicitly named, the default identity action will be used, which returns its input mentions unmodified (i.e. the input’s identity). Actions receive as input parameters this Mention sequence and also the runtime engine’s State.

The State object (second parameter) provides read-only access to all the mentions previously created by ODIN in the current document. This information may be useful to implement global decisions, e.g., coreference resolution across the entire document.

Actions must return a Mention sequence that will be added to the State at the end of the current iteration by the runtime engine. For example, the simplest
possible action would return the `mentions` it received as the first input parameter. Listing C.2 shows an only slightly more complicated action that removes any `Mention` containing the text “Fox”.

```scala
1 def action(mentions: Seq[Mention], state: State): Seq[Mention] = {
2   mentions.filter(_.text contains "Fox")
3 }
```

Listing C.2: An example of an action that removes any `Mention` containing the text “Fox”.

Note that, in addition to attaching actions to individual rules, actions can also be called globally at the end of each iteration by the runtime engine. This means that the extractor engine (see Listing C.3) must receive this global action as a parameter during its construction.

```scala
// The simplest instantiation where no actions are specified.
// Here the matches produced by our rules are returned unmodified.
val eeNoActions = ExtractorEngine(rules)
// myActions is an object containing the implementation of any actions
// named in the rules
val eeWithActions = ExtractorEngine(rules, myActions)
// Here we specify both an actions object and a global action
val eeWithActionsAndGlobal = ExtractorEngine(rules, myActions, myGlobalAction)
// We can also choose to specify only a global action
val eeWithGlobalOnly = ExtractorEngine(rules, globalAction = myGlobalAction)
```

Listing C.3: The `ExtractorEngine` may be instantiated in several ways.

Global actions have the same signature, but, in this context, the `mentions` parameter contains all mentions found in this iteration of the engine. Any mentions produced by rule-local actions will only make it into the `State` iff they pass successfully through the global actions. By default, the global action returns its input unmodified (i.e. the input’s identity).
REFERENCES


