The Sigma NLP Manual
A Guide for Users and Developers

Adam Pease

April 28, 2018

Contents

List of Figures

Chapter 1

Preface

1.1 Introduction

SigmaNLP is a collection of tools for exploring language to logic translation. It integrates the Stanford CoreNLP system with Sigma (Pease and Benzmller, 2013) and SUMO (Niles and Pease, 2001; Pease, 2011). CoreNLP handles the syntactic analysis of language and SigmaNLP addresses semantics. SigmaNLP uses Sigma as a component, so the SigmaNLP user and developer should first become familiar with Sigma.[1].

1.2 Code Overview

SigmaNLP has several packages under `com.articulate.nlp`. They are:

- `brat` at the moment contains one class to interface with the brat visualization component[2](Stenetorp et al., 2012). Brat is used to show dependency parse graphs in the JSP pages for SigmaNLP.
- `constants` contains some common linguistic constant terms and is not widely used
- `corpora` contains classes for the reading and evaluation of various linguistic corpora
- `lucene` has classes that interface with Lucene for basic information retrieval functions on text
- `pipeline` has utilities for running and displaying results from the linguistic processing pipeline of Stanford CoreNLP

[1] `https://gihub.com/ontologyportal/sigmakee`
[2] `http://brat.nlplab.org/`

- `semconcor` holds classes for the Semantic Concordancer (Pease and Cheung, 2018) that searches for semantic patterns in text
- `semRewrite` has all the classes of the Semantic Rewriting system that converts from dependency parses into logic statements in SUO-KIF using SUMO terms

The classes at the top level package `com.articulate.nlp` are a diverse collection of some machine learning experiments, CoreNLP Annotators and other utilities.

Chapter 2

CoreNLP

2.1 Pipeline

CoreNLP works on the principle of a *pipeline* of procedures, each adding annotations to an input text, and passing their annotations on to the next procedure. We use many of the available annotation procedures available from Stanford, and then add several of our own. Each pipeline is configured dynamically by listing the keywords associated with the annotators in a single string of properties. We assign these in the `Pipeline` object.

```
public static final String defaultProp = "tokenize, ssplit, " +
    "pos, lemma, ner, nersumo, gender, parse, coref, " +
    "depparse, wnmw, wsd, tsumo";
```

tokenize, ssplit, pos, lemma, ner, gender, parse, coref, and depparse are Stanford annotators. nersumo, wnmw, wsd, tsumo are added from SigmaNLP. The function of each of the annotators is as follows:

- `tokenize` separates a text into tokens, which are usually words. This largely means just separating words by spaces or punctuation, although it also means recognizing contractions to that "don't" turns into "do" and "n't".

- `ssplit` separates sentences. This requires some logic since things like periods, which separate sentences, also appear after honorifics like "Dr."

- `pos` is part of speech tagging. This means marking nouns and verbs as well as rarer parts of speech. It's easy to get to 90% accuracy just using the most frequent part of speech for a given word, but getting to higher accuracies is challenging. CoreNLP uses the Penn Treebank part of speech codes [1].

[1] `http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

- `lemma` annotates the root form of a word, which means removing things like plurals and past tense.
- `ner` annotates *named entities* like "United States" or "Arianna Grande". This is done with a machine learning algorithm trained on a hand-marked corpus. It recognizes features like punctuation and capitalization to make a guess at previously unseen terms.
- `nersumo` annotates named entities that are referred to in SUMO `termFormat` expressions and specifies the SUMO term they are equivalent to.
- `gender` annotates the gender of proper names, according to a database of common names.
- `parse` creates a parse tree, which organizes a sentence into a hierarchy of phrases and components.
- `coref` creates chains of linguistic reference. For example, Abraham Lincoln", "he", "the President", "The Great Emancipator" could all be used in a series of sentences, and must be linked as referring to the same person. `StanfordCorefSubstitutor` implements the function to select the best mention and substitute it for all the other references in the text.
- `depparse` creates a dependency parse graph.
- `wnmw` creates annotations of multiple word tokens from WordNet. These can be named entities like "United States" but also phrasal verbs like "look up". This is important since a phrase like "look up" has a specific meaning of the two words together that they don't have individually. `WNMultiWordAnnotator` implements this functionality.
- `wsd` is a Word Sense Disambiguation annotator. It attempts to determine which particular WordNet sense is being used in a sentence, based on statistics from the Brown corpus. It annotates words and phrases as to their SUMO type based on the WordNet annotation. `WSDAnnotator` implements this, based on the `WSD` and `WordNet` classes found in `com.articulate.sigma.wordNet` package in SigmaKEE.
- `tsumo` creates binary relation statements from the Stanford SUtime component that are more easily transformed into SUMO temporal expressions. `TimeSUMOAnnotator` implements this.

2.2 Adding an Annotator

The process of adding an annotator to the Stanford CoreNLP pipeline is fairly simple, but not completely described in the current documentation. This page provides the "hello world" solution for creating a new annotator.

You'll need to create two classes - the first is the actual annotator. An example is the class in the code shown below, called NEWAnnotator. It will be

passed the annotations so far, and have an opportunity to add more, and then pass the results on to the next annotator. The second class needed is really just a label that identifies the type of the annotation. It can be an interior class. In the code below, this is NEWAnnotation.

The annotator needs a constructor with a fixed signature consisting of a name and properties arguments. It must have three other methods - annotate(), requires() and requirementsSatisfied(). Each annotator in the pipeline, except the first, has requirements that must be satisfied before it can run, and requirements in turn that it satisfies for others. So we have to specify a sort of label for our new annotator with

```
static final Annotator.Requirement NEW_REQUIREMENT =
    new Annotator.Requirement("new");
```

and that is used in its requirementsSatisfied() method

```
@Override
public Set requirementsSatisfied() {
    return Collections.singleton(NEW_REQUIREMENT);
}
```

We also need to specify the requirements for our new annotator with the

```
public Set requires()
```

which returns a set of Annotator.Requirement. We'll also need to register this new annotator as part of the pipeline instance where it is used. To accomplish that, we do the following

```
Properties props = new Properties();
props.put("customAnnotatorClass.new","nlp.NEWAnnotator");
String propString =  "tokenize, ssplit, new";
Pipeline p = new Pipeline(true,propString);
```

We register the property as a "customAnnotatorClass" and put a string identifier for our new annotator after that. It can be any string. We then provide the full package and class name of our annotator as the property value, in our case "nlp.NEWAnnotator". Since we've tied the string identifier "new" to our class, we then use the identifier in the property string for the list of annotators in the pipeline object we create next "tokenize, ssplit, new". That's all there is to it! Now when we invoke the pipeline to annotate some text, as with

```
Annotation wholeDocument = p.annotate(contents);
```

our annotator will be called. If the "contents" parameter above is "Text for testing", we could use the following code to show the results

```
List sentences =
    annotation.get(CoreAnnotations.SentencesAnnotation.class);
for (CoreMap sentence : sentences) {
    List tokens =
        sentence.get(CoreAnnotations.TokensAnnotation.class);
```

```
    for (CoreLabel token : tokens) {
        String orig = token.originalText();
        String newAnno =
            token.get(NEWAnnotator.NEWAnnotation.class);
        System.out.println(orig + "/" + newAnno);
    }
}
```

We should get the output

```
Text/HERE for/HERE testing/HERE
```

The complete code for the annotator is given below.

```
package nlp;

import edu.stanford.nlp.pipeline.Annotation;
import edu.stanford.nlp.pipeline.Annotator;
import edu.stanford.nlp.pipeline.DefaultPaths;

import java.util.*;

import edu.stanford.nlp.ling.CoreAnnotation;
import edu.stanford.nlp.ling.CoreAnnotations;
import edu.stanford.nlp.ling.CoreLabel;
import edu.stanford.nlp.util.ArraySet;
import edu.stanford.nlp.util.CoreMap;

public class NEWAnnotator implements Annotator {

    public static class NEWAnnotation implements
        CoreAnnotation {
        public Class getType() {
            return String.class;
        }
    }

    static final Annotator.Requirement NEW_REQUIREMENT =
        new Annotator.Requirement("new");

    public NEWAnnotator(String name, Properties props) {

        // init here
    }

    public void annotate(Annotation annotation) {

        if (! annotation.containsKey(
            CoreAnnotations.SentencesAnnotation.class))
            throw new RuntimeException("Unable to " +
                "find sentences in " + annotation);
```

```
        List sentences =
            annotation.get(CoreAnnotations.SentencesAnnotation.class);
        for (CoreMap sentence : sentences) {
            List tokens =
                sentence.get(CoreAnnotations.TokensAnnotation.class);
            for (CoreLabel token : tokens) {
                token.set(NEWAnnotation.class, "HERE");
            }
        }
    }

    @Override
    public Set requires() {

        ArrayList al = new ArrayList<>();
        al.add(TOKENIZE_REQUIREMENT);
        al.add(SSPLIT_REQUIREMENT);
        ArraySet result = new ArraySet<>();
        result.addAll(al);
        return result;
    }

    @Override
    public Set requirementsSatisfied() {
        return Collections.singleton(NEW_REQUIREMENT);
    }
}
```

Chapter 3

Semantic Rewriting

Semantic Rewriting is a method of converting largely syntactic dependency parses into a deep semantic representation. The present work is an implementation of an approach by (Crouch, 2005; Crouch and King, 2006) performed within the Sigma Knowledge Engineering Environment (Pease and Benzmller, 2013), adapted to use the Stanford CoreNLP tools (Manning et al., 2014a), and employing the Suggested Upper Merged Ontology (Niles and Pease, 2001; Pease, 2011) as the semantics for the terms used in the ouput, and to interpret the semantics of words during the rewriting process.

The rule language used in sementic rewriting is given below in BNF.

```
Rule ::=    LHS ==> RHS.          Obligatory rewrite
            LHS ?=> RHS.          Optional rewrite

LHS ::=     Clause                Match & delete atomic clause
            +Clause               Match & preserve atomic clause
            LHS, LHS              Boolean conjunction
            (LHS | LHS)           Boolean disjunction
            -LHS                  Boolean negation
            {ProcedureCall}       Procedural attachment

RHS ::=     Clauses               Set of replacement clauses
            0                     Empty set of replacement clauses
            Stop                  Abandon the analysis

Clause ::=  Atom(Term,...,Term)   Clause with atomic predicate
            Atom                  Atomic clause
            Atom*                 match the word ignoring token number

Term ::=    Variable
            Clause
```

Classes in `com.articulate.nlp.semRewrite` implement an abstract syntax

tree for this grammar. It begins with `RuleSet` to contain the set of `Rule`. A `Rule` has an `LHS` and an `RHS`. Both an `LHS` and an `RHS` contain a list of `Clause` which in turn contain a list of `Literal`. Each `Literal` contains two of Stanford's `CoreLabel` for each argument, so that it can carry all the linguistic annotations that are performed. The `pred`icate in a `Literal` is the relationship between the two arguments. It can be a `Procedure` that needs to be run, as opposed to needing a literal match with the input.

A simple sentence like 1 results in a full logical expression. The logical output can be read as "There is a Man and Walking and a RetailStore such that the agent of the walking in the man and the destination of the walking is the RetailStore."

(1) $ Robert walks to the store.

```
(exists (?store-5 ?Robert-1 ?walk-2)
  (and
    (agent ?walk-2 ?Robert-1)
    (destination ?walk-2 ?store-5)
    (instance ?Robert-1 Man)
    (instance ?walk-2 Walking)
    (instance ?store-5 RetailStore)))
```

Now we can break down the steps to generate the logical output. We do not describe the process involving in parsing and dependency parsing, since that is done within the Stanford CoreNLP system, and the reader should refer to their publications for details.

We begin with the CoreNLP dependency parse of the sentence. Each literal consists of a binary dependency relation and two arguments, which are words from the input sentence and are annotated with the token number as suffix. We used the collapsed dependency format which converts prepositions into relations, rather than arguments. So, we see that there is no "to-3" in the output but rather the relation "prep_to" between "walks-2" and "store-5".

```
[root(ROOT-0, walks-2), nsubj(walks-2, Robert-1), det(store-5, the-4),
prep_to(walks-2, store-5)]
```

The next step is to apply the Word Sense Disambiguation process in the Sigma Knowledge Engineering Environment to guess at the right senses of any polysemous words, and find their corresponding SUMO terms, using the SUMO-WordNet mappings. In the case of this sentence, there is very little context to help determine the word senses so the system defaults to using the most frequent sense in WordNet SemCor. This results in adding pseudo-dependency literals with the relation "sumo". `WSDAnnotator` implements this, based on the `WSD` and `WordNet` classes found in `com.articulate.sigma.wordNet` package in SigmaKEE.

```
[root(ROOT-0, walks-2), nsubj(walks-2, Robert-1), det(store-5, the-4),
prep_to(walks-2, store-5), sumo(RetailStore,store-5),
sumo(Walking,walks-2), sumo(Man,Robert-1)]
```

Dependency parses do not provide parts of speech, or word morphology, which are determined in the previous phase in the CoreNLP pipeline. We reduce words to their part of speech and create more pseudo-dependency literals to hold information about tense, number and gender.

```
sumo(Man,Robert-1), number(SINGULAR, Robert-1), root(ROOT-0, walk-2),
nsubj(walk-2, Robert-1), sumo(Walking,walk-2), tense(PRESENT, walk-2),
sumo(RetailStore,store-5), number(SINGULAR, store-5),
prep_to(walk-2, store-5), det(store-5, the-4)
```

At this point we've completed preprocessing and it's time to apply semantic rewriting. This is a simple rule based system that test rules in a file in lexical order, applying them when possible then moving on to test the next rule in sequence. The system keeps making passes through the set of rules until no more rules can fire.

```
prep_to(?X,?Y), +sumo(?C,?Y), isCELTclass(?C,Object) ==>
  (destination(?X,?Y)). ; line 536
```

This first rule, which the system tells us occurs on line 536 for easy reference, to apply matches the `prep_to(walk-2, store-5)` and `sumo(RetailStore,store-5)` literals. It also calls a procedure `isCELTclass` to test the type of the second argument to `prep_to`. CELT was an earlier system that converted natural language to logic and it had some "informal" linguistic classes that are related to SUMO classes, but not identical with them. There are three classes of this sort - "Person", "Time" and "Object", which correspond to SUMO `Human` and `SocialRole`, `TimeMeasure` and `Process` and everything else, respectively.

The matching rule generates a new literal of the form `(destination(?X,?Y))`. The rule matching process performs *unification*, which means finding an instantiation of variables that is consistent. For this rule and input, we can see that `[?X=walk-2,?Y=store-5, ?C=RetailStore]` . This makes the rule

```
prep_to(walk-2, store-5), +sumo(RetailStore,store-5),
isCELTclass(RetailStore,Object) ==>
  (destination(walk-2,store-5)).
```

Rules "consume" the literals they match. After this rule matches, the literal `prep_to(walk-2, store-5)` is deleted and no longer available to match the left hand side of other rules. We use the "+" sign to allow us to match and keep a literal, so the literal `sumo(RetailStore,store-5)` is not deleted, and it remains available for use later. Since only the `prep_to(walk-2, store-5)` literal has been removed, and we've added `destination(walk-2,store-5)` literal, our available set of literals is

```
sumo(Man,Robert-1), number(SINGULAR, Robert-1), root(ROOT-0, walk-2),
nsubj(walk-2, Robert-1), sumo(Walking,walk-2), tense(PRESENT, walk-2),
sumo(RetailStore,store-5), number(SINGULAR, store-5),
det(store-5, the-4), destination(walk-2,store-5)
```

The next rule to apply is

```
nsubj(?E,?X), +sumo(?A,?E), isSubclass(?A,Process),
+sumo(?C,?X), isSubclass(?C,Agent) ==> (agent(?E,?X)). ; line 1014
```

This rule shows the use of another procedure call `isSubclass`, which checks to see if the given class is the subclass of another SUMO class. We also have the call `isInstanceOf` to check the class membership of instances.

```
root(?X,?Y) ==> !. ; line 1078
```

This rule says simply to delete the `root` literal.

```
agent(?X,?Y) ==> {(agent ?X ?Y)}. ; line 1198

destination(?X,?Y) ==> {(destination ?X ?Y)}. ; line 1204

sumo(?T,?X) ==> {(instance ?X ?T)}. ; line 1261
```

The curly braces around the right hand side of the rule signify that these are not dependency clauses and therefore they are part of the final logical output, and cannot be used in matching future rules. At this point the system has done a single pass through the entire list of rules. Several literals have not been consumed in the process:

```
[number(SINGULAR,Robert-1), sumo(Walking,walk-2), tense(PRESENT,walk-2),
sumo(RetailStore,store-5), number(SINGULAR,store-5), det(store-5,the-4)]
```

So, the system begins again, trying to match rules, starting at the beginning of the rule file.

```
sumo(?T,?X) ==> {(instance ?X ?T)}. ; line 1261
```

It finds just one match in the file, and winds up with the following literal set.

```
[number(SINGULAR,Robert-1), tense(PRESENT,walk-2),
sumo(RetailStore,store-5), number(SINGULAR,store-5), det(store-5,the-4)]

sumo(?T,?X) ==> {(instance ?X ?T)}. ; line 1261
```

Again just one rule matches.

```
[number(SINGULAR,Robert-1), tense(PRESENT,walk-2),
number(SINGULAR,store-5), det(store-5,the-4)]

[]
```

Finally, no rules match and therefore nothing more can be done, even though some clauses remain. The process ends. The set of logical clauses are assembled into a conjunction, and any variables are added to an existential quantifer list.

```
(exists (?store-5 ?Robert-1 ?walk-2)
  (and
    (agent ?walk-2 ?Robert-1)
    (destination ?walk-2 ?store-5)
    (instance ?Robert-1 Man)
    (instance ?walk-2 Walking)
    (instance ?store-5 RetailStore)))
```

The primary class in Semantic Rewriting is `Interpreter`. Running the help command for the interpreter (with your correct paths)

```
\$ java -Xmx2500m -classpath /home/apease/workspace/sigmakee/build/classes:
/home/apease/workspace/sigmakee/build/lib/*
com.articulate.sigma.semRewrite.Interpreter -h
```

gives the following output

```
Semantic Rewriting with SUMO, Sigma and E
  options:
  -h - show this help screen
  -s - runs one conversion of one sentence
  -i - runs a loop of conversions of one sentence at a time,
       prompting the user for more.  Empty line to exit.
       'load filename' will load a specified rewriting rule set.
       'ir/autoir/noir' will determine whether TF/IDF is run always,
           on inference failure or never.
       'reload' (no quotes) will reload the rewriting rule set.
       'inference/noinference' will turn on/off inference.
       'sim/nosim' will turn on/off similarity flooding (and
           toggle inference).
       'addUnprocessed/noUnprocessed' will add/not add
           unprocessed clauses.
       'showr/noshowr' will show/not show what rules get matched.
       'showrhs/noshowrhs' will show/not show what right hand
           sides get asserted.
       'quit' to quit
       Ending a sentence with a question mark will trigger a query,
       otherwise results will be asserted to the KB. Don't end
           commands with a period.
```

To make changes to the rewriting rules you can edit SemRewrite.txt . You can verify where that file is in your system by checking the output when the interpreter is run. You should see a line like

```
INFO in Interpreter.loadRules(): 421 rules loaded from
/home/apease/workspace/sumo/WordNetMappings/SemRewrite.txt
```

Try using the example input sentence described above and change the rule at line 536 to use the relation 'foo' instead of 'destination'. You can either type 'reload' to reload the file if you're in the interactive mode of the interpreter, or just rerun the entire command, as with

```
\$ java -Xmx2500m -classpath /home/apease/workspace/sigmanlp/build/classes:
/home/apease/workspace/sigmanlp/build/lib/*
com.articulate.sigma.semRewrite.Interpreter -i
```

3.1 JSP Interface

SigmaNLP has a number of functions controlled through its JSP-based interface. SigmaNLP, like Sigma, runs under Tomcat, which is a web server that allows Java code to be embedded in HTML pages and run on the server, as opposed to JavaScript, which is run on the client. Running SigmaNLP depends upon running Sigma

- `NLP.jsp` This is the primary page for running Semantic Rewriting. It has a text box for providing a sentence or sentences and then sections that show several stages of analysis as well as the final logical output. It calls `Interpreter` to do language to logic translation but also calls intermediate stages of the pipeline in order to present those results separately. It also calls the Brat vizualization component to show the augmented dependency parse graph. It calls `TimeBank` to generate canonicalized time expressions and gets the results as a `Timex` instance from the `TimeAnnotations` and `TimeSUMOAnnotator`. It calls the `RelExtract` relation extraction component. Word senses and multi-word phrases are retrieved from the `WSDAnnotator` and `WNMultiWordAnnotator`, respectively.

- `semconcor` This page is an interface to the semantic concordancer, which is handled in two classes `Indexer` and `Searcher`. It allows the user to search for a string or for a dependency parse fragment. If a result is found, it will show the words (for a search string) or literals (for a dependency parse fragment) that match in the found sentences. It also shows a Brat visualization.

- `unify.jsp` This function attempts to find a common pattern among two or more sentences. It calls `CommonCNFUtil`. The patten then can be searched for in the Semantic Concordancer.

- `SRrules.jsp` This page displays the `firedRules` member variable from `Interpreter`.

Chapter 4

Relation Extraction

We have a process for extracting relations from text. We utilize the over 1600 formally defined relations in the Suggested Upper Merged Ontology (SUMO) as the target relation set. Linguistic patterns to match in free text are determined by taking the natural language generation templates on SUMO relations and turning them into dependency parses using Stanford's CoreNLP system. We evaluate some of the results using a portion of the CoNLL04 corpus and provide some discussion about use of the corpus.

4.1 Introduction

Previous relation extraction work can be broadly divided into two types. In one, a small number, on the order of a dozen, of pre-defined relations are identified in text. In the other, a large number, from hundreds to millions, of relations are automatically determined from the text itself. Another way to group related research is into hand-built patterns (Grishman and Sterling, 1993; Hearst, 1992) vs. machine learning approaches (which in turn may be either clustering approaches or based on learning from training data).

In this present work, we rely on an existing corpus of hand-built relations in the Suggested Upper Merged Ontology(SUMO) (Niles and Pease, 2001; Pease, 2011)[1]. Unlike other sets of manually created relations, SUMO has definitions in higher order logic (HOL). The definitions provide guidance to both human and machine for what the relations mean and how and when to apply them. An automated theorem prover (Robinson and Voronkov, 2001) can mechanically determine the consistency of the use of these relations with respect to SUMO and any additional domain knowledge provided. This provides a more computable approach than, for example, WordNet's 22 semantic links (Fellbaum, 1998) or the four qualia relations in the generative lexicon (Pustejovsky, 1991), in which the meaning must involve an appeal to the training of a human linguist for interpretation.

[1] `http://www.ontologyportal.org`

Because our inventory of relation has already been built, our set is much larger than any new set of hand-built relations might be. As a logically-defined inventory, we do not have the problem that automated approaches to relation recognition do in having the same semantic relation appearing as several different names.

4.2 Pattern Generation

The SUMO project began in the year 2000 and many relations have been defined over the following 17 years. Along with most relations and their definitions, the ontology authors provide natural language generation templates, which were designed to improve the readability of the ontology by non-logicians. Our approach for this paper is essentially to run natural language generation "backwards", converting sample sentences into dependency parse patterns that can then be matched in free text.

In previous work on language to logic translation (Pease and Li, 2010) we developed what we might call "default" patterns that are applicable to many linguistic forms, such as that any subject of a sentence that has agency (member of the SUMO class `Agent`) gets the `agent` relationship to the action in the sentence. The patient relation expresses that the entity is the object of the action, and this is the default relation for the linguistic object of the sentence. For example, "Robert kicks the cart." becomes (paraphrased) "There is a kicking and a cart such that Robert is the agent of the kicking and the cart is the object of the kicking." This is more formally expressed in SUMO terms as

```
(exists (?K ?C)
  (and
    (instance ?K Kicking)
    (instance ?C Wagon)
    (agent ?K Robert)
    (patient ?K ?C)))
```

More complicated sentences can include temporal or modal constructs that require higher order logic (Benzmüller, 2015) in their resulting logical forms, and require the use of Discourse Representation Structures (Kamp and Reyle, 1993) during processing.

As the goal of this earlier work (called the Controlled English to Logic Translation system (Pease and Li, 2010)), and its more recent incarnation of Semantic Rewriting, is as full a logical expression of the semantics of an entire sentence as possible, it was necessary, at least initially, to have a very broad set of transformation rules that could generate a logical form for as many sentences as possible. In the present work, our goal is somewhat different in that we aim only to extract a small portion of the knowledge expressed in arbitrary sentences - just the relationships between entities, although we expect in a later stage to also apply it to interpretation of entire sentences. Because we want only to express relationships, we can expand our focus to look at a wide variety of relationship,

many with very detailed and specific semantics, far more narrow than notions of agency (or telicity etc).

Relation Extraction Rules

We generate an initial set of relation extraction rules automatically. The overall process is to use existing templates for natural language generation to create sample sentences, then use Stanford's CoreNLP system to turn them into dependency parses, then process the linguistic arguments into variables and use that as the left hand side for a rule.

As an example, one relatively specialized SUMO relation from among its set of 1606 relations is `engineeringSubcomponent`[2]. This is a relation between two things, which are both `EngineeringComponents` as defined with the following:

```
(domain engineeringSubcomponent 1 EngineeringComponent)
(domain engineeringSubcomponent 2 EngineeringComponent)
(instance engineeringSubcomponent BinaryPredicate)
```

For each relation that is created in SUMO, ontology authors are expected to include a natural language generation template, that allows the Sigma(Pease and Benzmller, 2013) system to generate natural language paraphrases for SUMO-based logical expressions. In the case of this relation it is

`(format EnglishLanguage engineeringSubcomponent "%1 is %n a component of %2")`

To create a fully expressed logical sentence with this relation, we add quantification for the arguments as follows

```
(exists (?E1 ?E2)
  (engineeringSubcomponent ?E1 ?E2))
```

Using the simple NLG approach in Sigma this results in

"there exist an engineering component and another engineering component such that the engineering component is a component of the other engineering component"

Running this through the CoreNLP system's dependency parser, and then augmenting the dependency parse with expressions for the SUMO-based types of words. This becomes the left hand side of a rule, to which we append a right hand side to generate the SUMO relation. This results in

```
nmod:of(component*,?component-11),
nsubj(component*,?component-3),
cop(component*,is*),
case(?component-11,of*),
sumo(EngineeringComponent,?component-11),
sumo(EngineeringComponent,?component-3) ==>
    engineeringSubcomponent(?component-3,?component-11).
```

[2] see `http://sigma.ontologyportal.org:8080/sigma/Browse.jsp?kb=SUMO&lang=EnglishLanguage&flang=SUO-KIF&term=engineeringSubcomponent` for the full definition of this term in SUMO

We should emphasize that these are linguistic matching rules, that serve a different purpose, and use a simple production rule formalism, rather than the higher order logic language used in SUMO itself.

As an example of the application of this rule, we can look at a sentence in Wikipedia

> *The display is the most expensive component of the OLPC Laptop.*

which winds up automatically generating, with associated type definitions

```
(and
  (instance ?D ComputerDisplay)
  (instance OLPC-Laptop Laptop)
  (engineeringSubcomponent ?D OLPC-Laptop))
```

4.3 NLP Pipeline

We utilize the Stanford CoreNLP Manning et al. (2014b) pipeline and augment it with additional linguistic annotators that take advantage of information in SUMO and WordNet, and the SUMO-WordNet mappings Niles and Pease (2003). We perform a simple approach to Word Sense Disambiguation based on the data in WordNet SemCor Landes et al. (1998) that identifies WordNet word senses and then uses the SUMO-WordNet mappings to find SUMO terms that are equivalents or near-equivalents to each word in the sentence.

After CoreNLP has performed Named Entity Recognition and Dependency Parsing, we attempt to refine the types of terms in a sentence. Priority is given to concepts in SUMO that are rendered in English as Multi-Word Expressions (MWEs), such as "United Nations Development Program" [3]. Then MWEs in WordNet are the next priority. In each case, we use a "greedy" algorithm that just matches MWEs as aggressively as possible. For example, "John kicked the bucket." will always match the idiom for "to die" rather than the literal sense of the phrase. If both those sources fail, we then look at Stanford's NER types, and its proper name annotator results. Stanford will make a guess, not based on any corpus of words or phrases, on the meaning of things like "NVidia" or "Guadalupe Parkway". It's common name processor will find the gender of common Western names, like "Jessica".

Once the NLP pipeline has completed, then relation extraction rules can be applied to the augmented dependency parses.

[3] https://nlp.ontologyportal.org:8443/sigma/Browse.jsp?lang=EnglishLanguage&flang=SUO-KIF&kb=SUMO&term=UnitedNationsDevelopmentProgram

4.4 Corpus

We have attempted to use the CoNLL04 relation corpus [4]. The class to handle this corpus is `com.articulate.nlp.corporate.CoNLL04`. It is a corpus of 5926 sentences, roughly 1400 of which have some marked relation. Relations we marked by hand. Five relations are specified, which are: located in, work for, organization based in, live in, and kill. We had some existing relations defined in SUMO, which at least at first appeared to match. These correspond to `located`, `employs` (also `leader`), `located` (when the first argument is a `Organization`), and `inhabits` or `birthplace`. SUMO needs a relation for 'kill', which is currently handled as a type of action and roles for participants.

The format of the file is

- a sentence in table format
- empty line
- relation descriptors (may be empty or more than one line)
- empty line
- In the table of a sentence, each row represents an element (a single word or consecutive words) in the sentence. Meaningful columns include:
  - col-1: sentence number
  - col-2: Entity class label (B-Unknown, B-Peop, or B-Loc, which means, respectively, other_entity, person, location)
  - col-3: Element order number
  - col-5: Part-of-speech tags
  - col-6: Words

A relation descriptor has three fields.

- 1st field : the element number of the first argument.
- 2nd field : the element number of the second argument.
- 3rd field : the name of the relation (e.g. kill or birthplace).

Note that this corpus, despite all the publications associated with it, has some significant issues, with sentences apparently marked with recourse to world knowledge rather than the strict content of the sentence, such as "Booth shot Lincoln" being marked as a "killed" relation.

[4] `http://cogcomp.org/page/resource_view/43`

Bibliography

Benzmüller, C. (2015). Higher-order automated theorem provers. In Delahaye, D. and Woltzenlogel Paleo, B., editors, *All about Proofs, Proof for All*, Mathematical Logic and Foundations, pages 171–214. College Publications, London, UK.

Crouch, R. (2005). Packed rewriting for mapping semantics to KR. In *Proc. 6 th Int. Workshop on Computational Semantics*, pages 103–114, Tilburg.

Crouch, R. and King, T. H. (2006). Semantics via f-structure rewriting. In *Proceedings of LFG06*, pages 145–165.

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. Language, Speech, and Communication. MIT Press.

Grishman, R. and Sterling, J. (1993). New york university: Description of the proteus system as used for muc-5. In *Proceedings of the 5th Conference on Message Understanding*, MUC5 '93, pages 181–194, Stroudsburg, PA, USA. Association for Computational Linguistics.

Hearst, M. A. (1992). Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the 14th Conference on Computational Linguistics - Volume 2*, COLING '92, pages 539–545, Stroudsburg, PA, USA. Association for Computational Linguistics.

Kamp, H. and Reyle, U. (1993). *From Discourse to Logic. Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory.* Kluwer, Dordrecht.

Landes, S., Leacock, C., and Tengi, R. (1998). Building semantic concordances. In Fellbaum, C., editor, *WordNet: An Electronic Lexical Database*, Language, Speech, and Communication. MIT Press, Cambridge (Mass.).

Manning, C., Bauer, J., Surdeanu, M., Finkel, J., Bethard, S., and McClosky, D. (2014a). The stanford corenlp natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. (2014b). The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60.

Niles, I. and Pease, A. (2001). Toward a Standard Upper Ontology. In Welty, C. and Smith, B., editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*.

Niles, I. and Pease, A. (2003). Linking Lexicons and Ontologies: Mapping WordNet to the Suggested Upper Merged Ontology. In *Proceedings of the IEEE International Conference on Information and Knowledge Engineering*, pages 412–416.

Pease, A. (2011). *Ontology: A Practical Guide*. Articulate Software Press, Angwin, CA.

Pease, A. and Benzmller, C. (2013). Sigma: An Integrated Development Environment for Logical Theories. *AI Communications*, 26:9–97.

Pease, A. and Cheung, A. K.-F. (2018). Toward a semantic concordancer. In Bond, F., Fellbaum, C., and Vossen, P., editors, *The 9th Global WordNet Conference*.

Pease, A. and Li, J. (2010). Controlled English to Logic Translation. In Poli, R., Healy, M., and Kameas, A., editors, *Theory and Applications of Ontology*. Springer.

Pustejovsky, J. (1991). The generative lexicon. *Comput. Linguist.*, 17(4):409–441.

Robinson, A. and Voronkov, A. (2001). *Handbook of Automated Reasoning*. MIT Press.

Stenetorp, P., Pyysalo, S., Topić, G., Ohta, T., Ananiadou, S., and Tsujii, J. (2012). Brat: A web-based tool for nlp-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, EACL '12, pages 102–107, Stroudsburg, PA, USA. Association for Computational Linguistics.

Index of Classes