Information Extraction System
Software Design Specification

By – Sanjay Meena

Revision history

Version	Editor	Date	Description
0.1	Sanjay Meena	28/9/2013	Started writing this SDS Document.
0.2	Sanjay Meena	30/9/2013	Finished version with Module details. More details need to be added to some modules

1 Table of Content

Contents

2 Table of Figures

3 Introduction

This is the Software Design Specification Document for Information Extraction System. It will have details related to the implementation side of Information Extraction System.

4 Document Description

4.1 Introduction

4.1.1 Describe the purpose of this document

This document describes the Detailed Architecture, Technologies for the Information Extraction System.

4.1.2 Describe this document's intended audience

Information Extraction System is a core system. Chinese information extraction system is also on similar lines. Anybody dealing with NLP in their Projects may require knowing about this system.

4.1.3 Provide references for any other pertinent documents such as:

4.1.3.1 Related and/or companion documents

- The Proposal on Information Extraction System
- Unified Knowledge Graph Representation
- Old Information Extraction and Question Generation System

4.1.3.2 Prerequisite documents

- Project Proposal for Information Extraction System
- Unified Knowledge Graph Representation

4.1.3.3 Documents which provide background and/or context for this document

- Information Extraction System

4.1.3.4 Documents that result from this document (e.g. a test plan or a development plan)

4.1.4 Define any important terms, acronyms, or abbreviations

- IE: Information Extraction
- NER: Named Entity Recognition

4.1.5 Summarize (or give an abstract for) the contents of this document.

This document provides the design details of information extraction system. Each module is discussed. Also insights and reasons for using each module is included.

4.2 System Overview

Dealing with Text, whether in the form of Article, Dialogues is an important and challenging task. Extracting structured information from the text is a fundamental step to build intelligent applications which directly or indirectly relates to human.

4.3 Design Considerations

Following design considerations are important:

- Follow Agile Method of Programming. Spend more time on quicker implementation rather than spending lot of time on design and documents. Development needs to be in iterations. Have a product and keep it improving then.
- The output from the IE system is language independent. It can be human readable representation (json, xml).
- Follow design patterns for programming. This will mean writing more code, but it ensures maintenance, modularity and easy enhancement of the System.
- The Modules in the system need to be really that, modular. They need to be able to work independently by themselves.
- Need to have unit test cases for each module for testing.
- Having a Test data to consistently keep testing and improving the system.

4.3.1 Assumptions and Dependencies

The IE system depends on some external modules on which independent research group have worked on. Following are the key dependencies of this system. Please note that for Chinese IE System, dependencies are different.

4.3.1.1 Dependency on existing NLP tools

1.1.1.1.1 Stanford Phrase Structure Parser

http://nlp.stanford.edu/software/lex-parser.shtml

Stanford parser is used to automatically sentence-split, tokenize, and parse input texts—resulting in Penn Treebank-style (Marcus et al., 1993) phrase structure syntax trees for each sentence.

4.3.1.1.1 The Tregex Tree Searching Language and Tool

http://nlp.stanford.edu/software/tregex.shtml

Tregex tree searching language (Levy and Andrew, 2006) can identify the relevant syntactic elements (e.g., subjects of sentences) as well as larger constructions (e.g., non-restrictive appositives) that our system operates on. Tregex allows us to match complex patterns in trees by combining various types of node-to-node relationships such as dominance and precedence and also nesting such relationships.

4.3.1.1.2 Supersense Tagger

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.178.7563

Supersense tagger Ciaramita and Altun (2006) is to label word tokens with high-level semantic classes from Wordnet (Miller et al., 1990), such as noun.person, noun.location , etc. The tagger labels proper nouns, as in the task of **named entity recognition** (Finkel et al., 2005), as well as common nouns and verbs. Supersense tagging can be viewed as a combination of named entity recognition, where just proper nouns are labeled with a small set of high-level semantic classes (often just PERSON, ORGANIZATION, and LOCATION), and word sense disambiguation (Yarowsky, 1995), where common nouns and verbs are labeled with very fine-grained word senses.

Consider an example: Steve jobs visited Taipei to give a presentation.

B-Noun.person	I-Noun. Person	B-verb.social	B-noun.location	0	B-verb.possession	0
Steve	Jobs	visited	Taipei	to	give	a
B-noun.act	0					
Presentation	.					

4.3.1.1.3 ArkRef Noun Phrase Coreference Tool

ArkRef is a tool for noun phrase Coreference that is based on the system described by Haghighi and Klein (2009). It is a deterministic system that uses syntactic information from the Stanford Parser (Klein and Manning, 2003) and semantic information from an entity recognition component to constrain the set of possible mention candidates (i.e., noun phrase nodes) that could be antecedents for a given mention. It encodes syntactic

constraints such as the fact that the noun phrases in predicative nom-

Example provides an illustration of ARKref's output, in which brackets denote the extent of noun phrases and indices denote the entity to which each noun phrase refers.

[John] $_1$ bought [himself] $_1$ [a book] $_2$. [Fred] $_3$ found out that [John] $_1$ had also bought [himself] $_1$ [a computer] $_4$. Next, [he] $_3$ found out that [John] $_1$ had not bought [him] $_3$ [anything] $_5$.

One can also interpret the output of a Coreference tool as an undirected graph with nodes for each noun phrase and edges between noun phrases that refer to the same entity.

4.3.1.1.4 English Wordnet

http://wordnet.princeton.edu/

English Wordnet was created and is being maintained at the Cognitive Science Laboratory of Princeton University under the direction of psychology professor George A. Miller. The development work began in 1989. Wordnet have been created in dozens of other languages since then.

4.3.1.2 Dependency on Resources

4.3.1.2.1 Play Framework

http://www.playframework.com/

To make the IE system or any Java based system available as a web service/ web app, Play Framework is a modern and scalable choice. Play is based on a lightweight, stateless, web-friendly architecture.

Built on Akka, Play provides predictable and minimal resource consumption (CPU, memory, threads) for highly-scalable applications.

Play was built for needs of modern web & mobile apps.

- RESTful by default
- Asset Compiler for CoffeeScript, LESS, etc
- JSON is a first class citizen
- Websockets, Comet, EventSource
- Extensive NoSQL & Big Data Support

Linkedin Klout The Guardian Gilt are already using play framework.

4.3.1.3 Operating System:

Windows/Linux/

4.3.1.3.1 For Android:

The IE system like most NLP systems requires good computation power and memory usage. Hence the most sensible choice seems to be to use the IE systems through a web service for systems like Android.

4.3.1.4 End-user characteristics:

Normally IE system does not have end user characteristic. IE system is meant to be serving applications which in turn serve the user.

4.3.1.5 Possible and/or probable changes in functionality

There might be changes in additions however the interfaces shall remain the same.

4.3.2 General Constraints

4.3.2.1 Hardware or software environment

4.3.2.1.1 Programming Language: Java

Output is language independent in various formats.

4.3.2.1.2 Hardware Requirements

CPU: Intel dual core 2.5 Ghz (or Equivalent) and higher

Minimum Memory requirement: 1.5 GB and higher

4.3.2.2 End-user environment

4.3.2.3 Availability or volatility of resources

4.3.2.4 Standards compliance

- Adherence to Object Oriented Programming Principles
- Adherence to Design Patterns in Programming

4.3.2.5 Interoperability requirements

The IE system at default level can run by itself. To be powerful, it requires to use

4.3.2.6 Interface/protocol requirements

JSON or XML format will be used for protocol. Both are very widely used.

4.3.2.7 Data repository and distribution requirements

- POSTGRES based Database to store the information generated from the IE system.

4.3.2.8 Security requirements (or other such regulations)

Currently, there are no security requirements

4.3.2.9 Memory and other capacity limitations

Minimum Memory requirement: 1.5 GB and higher

4.3.2.10 Performance requirements

4.3.2.11 Network communications

IE System will be available as a web service.

4.3.3 Goals and Guidelines

4.3.3.1 The KISS principle ("Keep it simple stupid!")

Although the systems are complex, the interfaces need to be as simple and intuitive as possible. The end user or even an intermediate programmer does not need to deal with the complexity of the system.

4.3.3.2 Program to an interface, not an implementation.

This principle is really about dependency relationships which have to be carefully managed in a large app. It's easy to add a dependency on a class. It's almost too easy; just add an import statement and modern Java development tools like Eclipse even write this statement for you. Interestingly the inverse isn't that easy and getting rid of an unwanted dependency can be real refactoring work or even worse, block us from reusing the code in another context. For this reason we have to develop with open eyes when it comes to introducing dependencies. This principle tells us that depending on an interface are often beneficial.

4.3.3.3 The value of interfaces

- An interface distills the collaboration between objects. An interface is free from implementation details, and it defines the vocabulary of the collaboration. Once we understand the interfaces, we understand most of the system.

4.3.3.4 Composition versus inheritance

Favor composition of classes rather than inheritance.

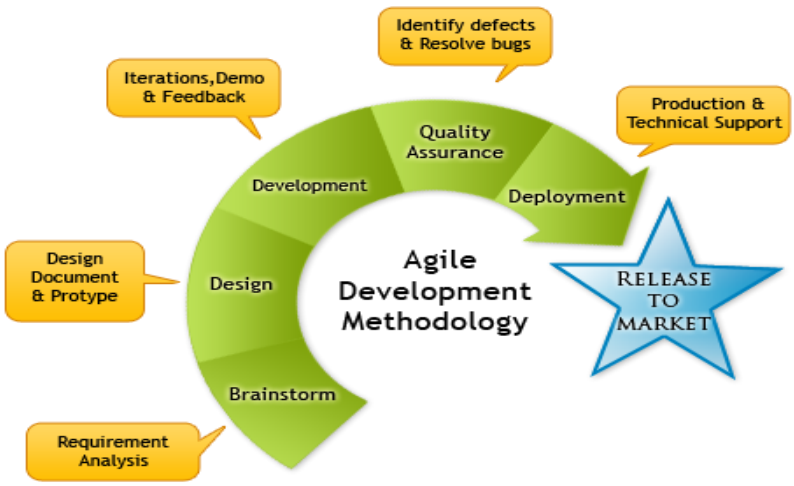
4.3.3.5 Development Methods

4.3.3.5.1 Agile Software Development

Agile software development is a group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. It promotes adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change. It is a conceptual framework that promotes foreseen interactions throughout the development cycle.

Some principles that constitute Agile methods are:

1. The reigning supreme of individuals and interactions over processes and tools.
2. As does, working software over comprehensive documentation.
3. Responding to change over plan follow-throughs.



4.3.3.5.2 Water Fall Model

The waterfall model is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation, and Maintenance.

4.4 Design Principles and Design Patterns

4.4.1 Object Oriented Programming Principles

OO programming suggests that we use the following principles during the design of software. The following are not "Design Principles" but a repetition of a good OO design.

4.4.1.1 Encapsulation

In general a general manipulation of an object's variables by other objects or classes is discouraged to ensure data encapsulation. A class should provide methods through which other objects could access variables. Java deletes objects which are no longer used (garbage collection).

4.4.1.2 Abstraction

Java supports the abstraction of data definition and concrete usage of this definition.

The concept is divided from the concrete which means you first define a class containing the variables and the behavior (methods) and afterwards you create the real objects which then all behave like the class defined it.

A class is the definition of the behavior and data. A class cannot be directly be used.

An object in an instance of this class and is the real object which can be worked with.

4.4.1.3 Polymorphisms

Polymorphism is the ability of object variables to contain objects of different classes. If class X1 is a subclass of class X then a method which is defined with a parameter for an object X can also get called which an object X1.

If we define a supertype for a group of classes any subclass of that supertype can be substituted where the supertype is expected.

If we use an interface as a polymorphic type any object which implements this interface can be used as arguments.

4.4.1.4 Inheritance

Inheritance allows that classes can be based on each other. If a class A inherits another class B this is called "class A extends class B".

For example you can define a base class which provides certain logging functionality and this class is extended by another class which adds email notification to the functionality.

4.4.1.5 Delegation

Delegation is then you hand over the responsibility for a particular task to another class or method.

If you need to use functionality in another class but you do not want to change that functionality then use delegation instead of inheritance.

4.4.1.6 Composition

When you refer to a whole family of behavior then you use composition. Here you program against an

interface and then any class which implements this interface can be used to be defined. In composition the composition class is still defined in the calling class.

When you use composition, the composing object owns the behaviors and they stop existing as soon as the composing object does.

4.4.1.7 Aggregation

Aggregation allows you to use behavior from another class without limiting the lifetime of those behaviors.

Aggregation is when one class is used as part of another class but still exists outside of that class.

4.4.1.8 Design by contract

Programming by contract assumes both sides in a transaction understand what actions generate what behavior and will abide by that contact.

Methods usually return null or unchecked exceptions when errors occur in programming by contract environment.

If you believe that a method should not get called in a certain way just throw an unchecked runtime exception. This can be really powerful. Instead of checking in your calling code for exceptions you just throw an exception in the called code. Therefore we can easily identify the place in the coding were an error occurs. This follows the "crash-early" principle, which tells that if an error occurs in your software you should crash immediately and not later in the program because then it is hard to find the error.

4.4.1.9 Cohesion

A system should have a high cohesion.

Cohesion is a measure of how strongly-related and focused the responsibilities of a single class are. In object-oriented programming, it is beneficial to assign responsibilities to classes in a way that keeps cohesion high.

Code readability and the likelihood of reuse is increased, while complexity is kept manageable, in a highly-cohesive system.

Therefore you should avoid classes which have several responsibilities, e.g. a Logger class should only be responsible for logging.

4.4.1.10 The Principle of Least Knowledge

Talk only to your immediate friends. It is also known as Law of Demeter. When you are designing a system, for any object be careful of the number of classes it interacts with and also how it interacts with those classes. This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts. When we build a lot of dependencies between many classes, you are building a fragile system that is costly to maintain and complex for others to understand.

Take any object. Now from any method in that object, we should only invoke methods that belong to:

1. The Object itself

2. Objects passed in as a parameter to the methods

3. Any of object that a method creates and instantiates.

4. Any components of objects. (Has-A relationship)

4.4.1.11 The Open Closed Principle

Software entities like classes, modules and functions should be open for extension but closed for modifications.

This principle encourages developers to write code that can be easily extended with only minimal or no changes to existing code.

An example for a good application of these principles would be that a certain class calls internally an abstract class to conduct a certain behavior. At runtime this class is provided with a concrete implementation of this abstract class. This allows the developer later to implement other concrete calls of this abstract class without changing the code of the class which uses this abstract class.

4.4.2 Design Patterns

Design Patterns are proven solutions approaches to specific problems. A design pattern is not framework and is not directly deployed via code.

Design Pattern has two main usages:

- **Common language for developers**: They provide developer a common language for certain problems. For example if a developer tells another developer that he is using a Singleton, the developer (should) know exactly what this means.

- **Capture best practices**: Design patterns capture solutions which have been applied to certain problems. By learning these patterns and the problem they are trying to solve. An inexperienced developer can learn a lot about software design.

4.4.2.1 Some Important Design Patterns

Here only brief description is provided; to keep the size of this document small as Design patterns needs a separate discussion.

4.4.2.1.1 Singleton Pattern

A singleton in Java is a class for which only one instance can be created provides a global point of access this instance. The singleton pattern describes how this can be archived.

Singletons are useful to provide a unique source of data or functionality to other Java Objects.

4.4.2.1.1.1 For example

You may use a singleton to access your data model from within your application or to define logger which the rest of the application can use.

4.4.2.1.2 Observer Pattern

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

1. Example

The observer pattern is very common in Java. For example you can define a listener for a button in a user interface. If the button is selected the listener is notified and performs a certain action.

4.4.2.1.3 Facade Pattern

Facade exposes a simplified interface (in this case a single interface to perform that multi-step process) and internally it interacts with those components and gets the job done for you. It can be taken as one level of abstraction over an existing layer.

2. Example

Let's take a car; starting a car involves multiple steps. Imagine how it would be if you had to adjust n number of valves and controllers. The facade you have got is just a key hole. On turn of a key it sends instruction to multiple subsystems and executes a sequence of operation and completes the objective. All you know is a key turn which acts as a facade and simplifies your job.

4.4.2.1.4 Adaptar Pattern

An adapter helps two incompatible interfaces to work together. This is the real world definition for an adapter. Adapter design pattern is used when you want two different classes with incompatible interfaces to work together. The name says it all. Interfaces may be incompatible but the inner functionality should suit the need.

3. Example

In real world the easy and simple example that comes to mind for an adapter is the travel power adapter. American socket and plug are different from British. Their interfaces are not compatible with one another. British plugs are cylindrical and American plugs are rectangularity. We can use an adapter in between to fit an American (rectangular) plug in British (cylindrical) socket assuming voltage requirements are met with.

4.4.2.1.5 Decorator Design Pattern

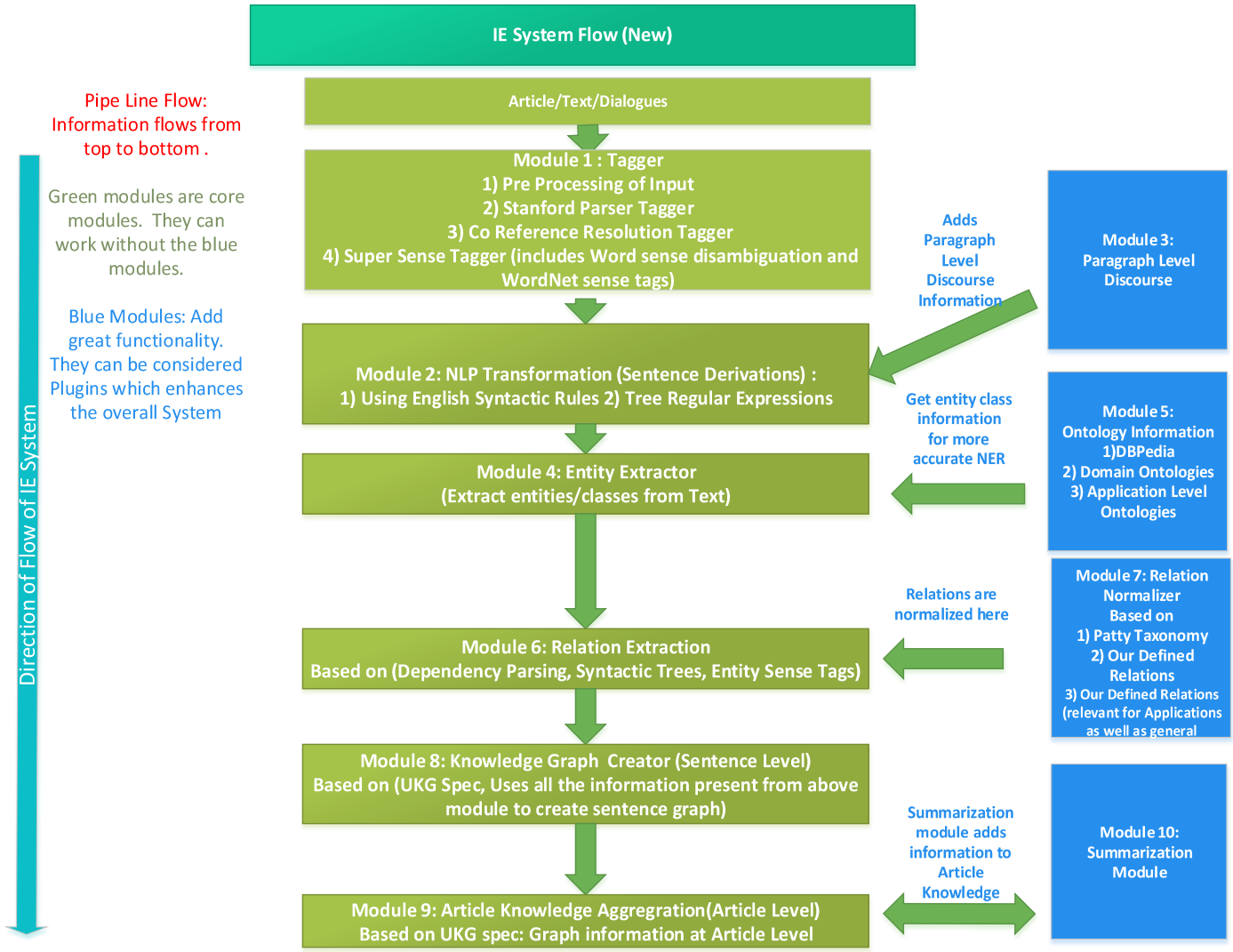
In object-oriented programming, the **decorator pattern** is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.

This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s).

5 System Architecture

- IE systems follow the Pipe line flow of information. Every Module adds information and the modules below use the information added by the previous modules.
- There are currently 10 modules.
  - The green modules listed below are core modules. They form the core Information Extraction system.
  - Blue Modules are like plugin. They add very useful information and expands the functionality of the core system greatly

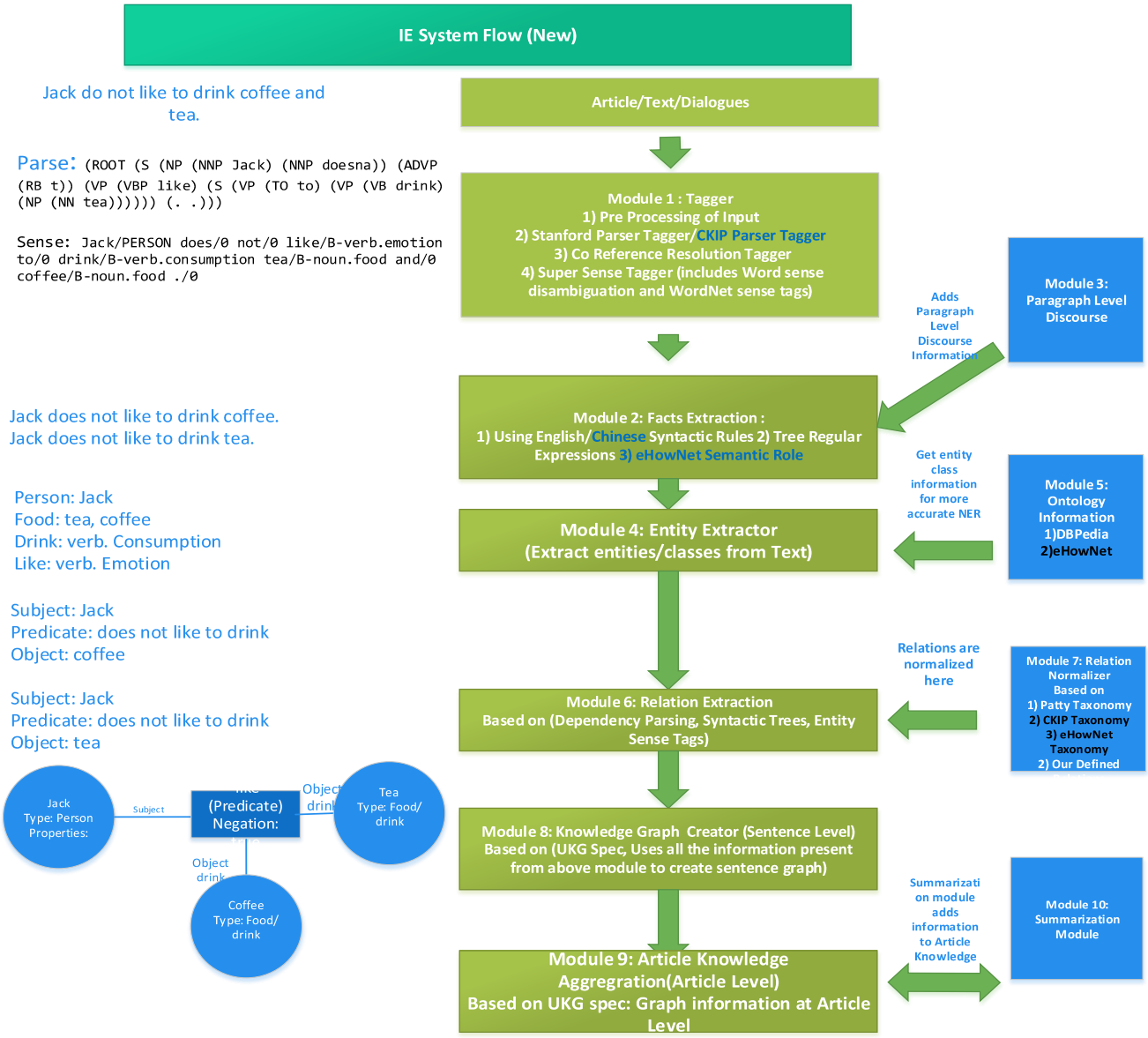
5.1 English IE System



5.2 Example of the System Flow

Below an example of the System flow is described. In the given

Figure 1: IE System Example Flow



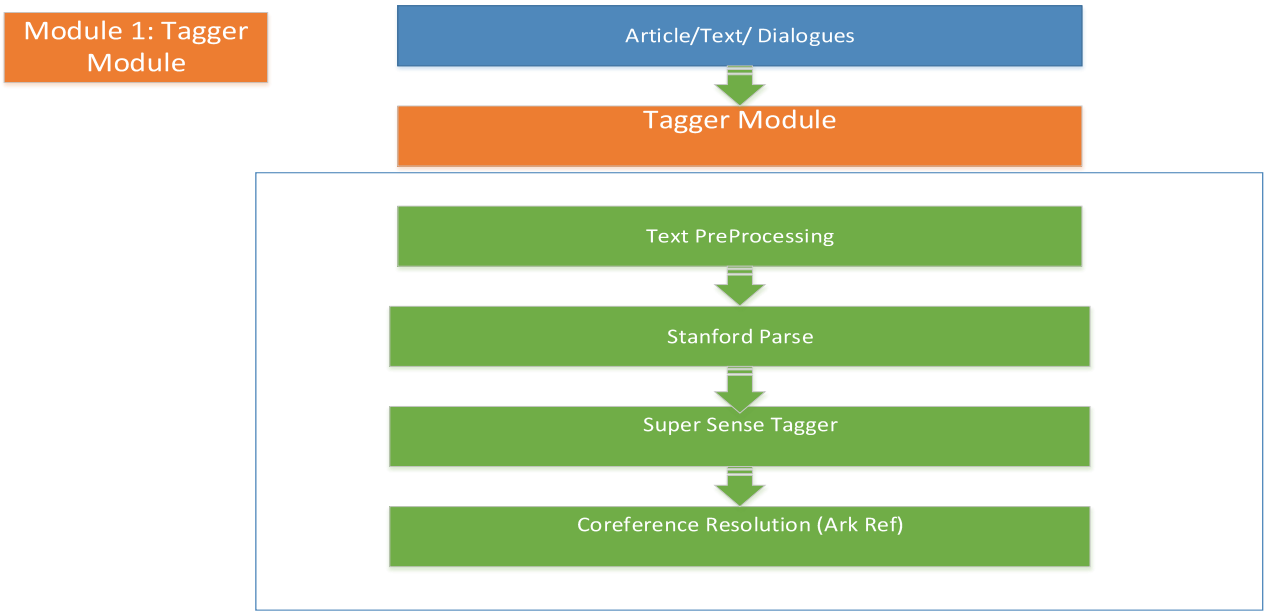
6 Subsystem Architecture

Here we discuss each module in detail

6.1 Module 1 : Tagger Module

Tagger Module is the first module of the IE System. This is the first module in the IE system. Let's discuss each part in some detail

Figure 2: Module 1: Tagger Module



Note: Module has a Pipeline Flow

6.1.1 Text Preprocessing

The input text can contain many deficiencies, which should be taken care before the actual tagging begins.
Examples:

- Replace encoding symbols , e.g. "é|è|ë|ê" to "e"
- Perform some transformations for better parsing:
  - Replace 'd –> Would, 'll -> will etc

The complete list of preprocessing are mentioned in the code

6.1.2 Stanford Parse

Stanford Parser can provide parse trees and dependency parses. This is the core component as the parse trees are used in almost every module.
Consider an example sentence:

```
I would like to watch a movie tonight in Taipei starring Tom Cruise.
```

6.1.2.1 Parse Tree:

```
(ROOT
  (S
```

```
    (NP (PRP I))
    (VP (MD would)
      (VP (VB like)
        (S
          (VP (TO to)
            (VP (VB watch)
              (NP
                (NP (DT a) (NN movie))
                (NP (NN tonight))
                (PP (IN in)
                  (NP (NNP Taipei)))
                (PP (VBG starring)
                  (NP (NNP Tom) (NNP Cruise)))))))))
    (. .)))
```

This is the core structure for NLP information and also to perform many operations to extract meaningful information.

6.1.2.2 Dependency Parse

```
nsubj(like-3, I-1)
xsubj(watch-5, I-1)
aux(like-3, would-2)
root(ROOT-0, like-3)
aux(watch-5, to-4)
xcomp(like-3, watch-5)
det(movie-7, a-6)
dobj(watch-5, movie-7)
iobj(watch-5, movie-7)
dobj(watch-5, tonight-8)
dep(movie-7, tonight-8)
prep_in(movie-7, Taipei-10)
nn(Cruise-13, Tom-12)
prep_starring(movie-7, Cruise-13)
```

Dependency Parse has its use in Relation Extraction. It should be used in conjunction with parse tree for more detailed relation extraction.

6.1.3 SuperSense Tagger Module

- This module tags the sense information with each word. This sense information is retrieved from Wordnet.
- Word Sense Disambiguation is also performed as this module knows the POS tag as well the neighborhood information of each word.
- The input is the Stanford part of speech information from the Stanford Parser Module.

```
Sentence: I would like to watch a movie tonight in Taipei starring Tom Cruise.

Supersense: I/0 would/0 like/B-verb.emotion to/0 watch/B-verb.perception a/0 movie/B-
noun.communication tonight/B-noun.time in/0 Taipei/B-noun.location starring/B-
verb.stative Tom/PERSON Cruise/PERSON ./0
```

6.1.4 Coreference Resolution using ArkRef Coreference Tool

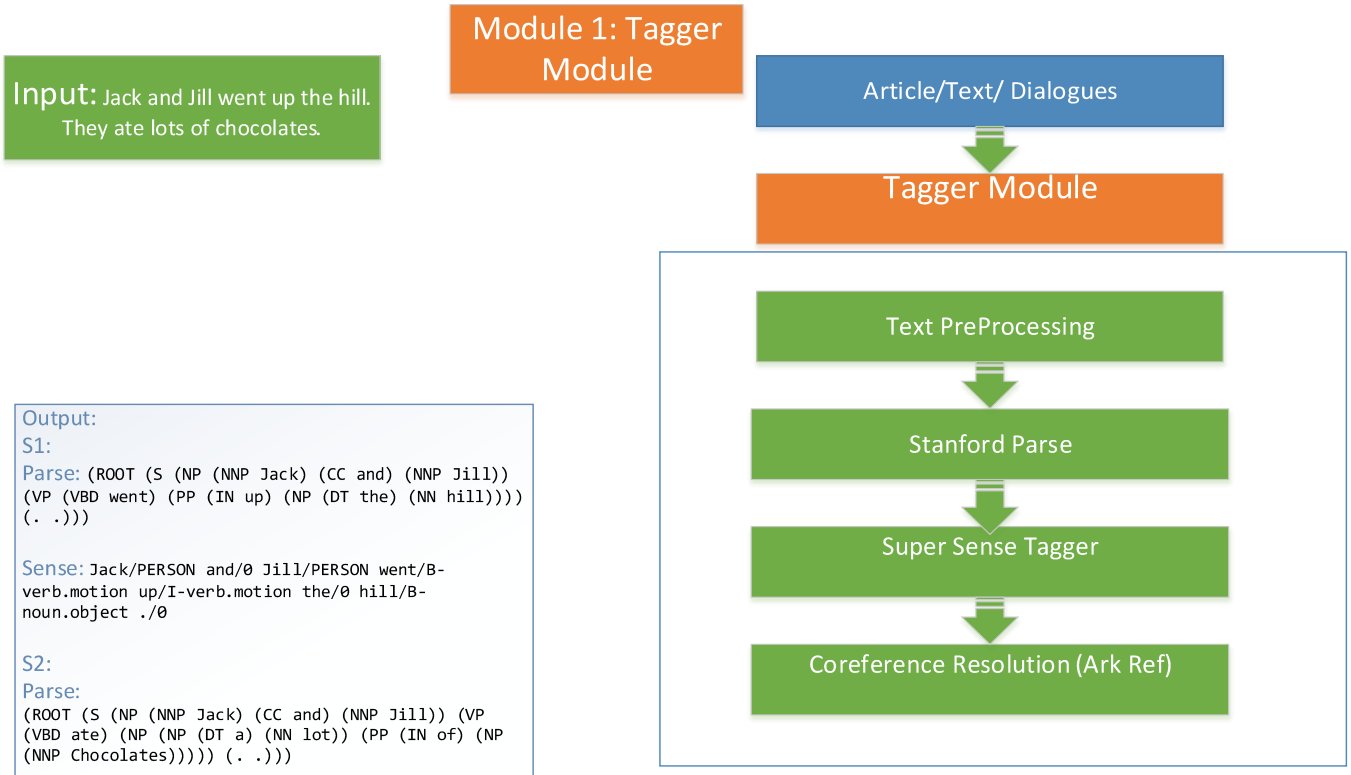
- This tool needs Stanford parse and Super sense tag information to be able to perform Coreference Resolution.
- The advantage is that sense information is used in conjunction with syntactic information for better coreferencing.

The input sentence containing co-referents are modified here to include the Coreference information. This module has its own data structure which is added to the System data structure.

Below is an Example of the Output:

6.1.5 Example Output

Figure 3: Example Output - Tagger Module



6.2 Module 2: Fact Extraction Module

- This module is a necessary module to break down the complexity of complex, compound sentences.
- This module performs various syntactic transformations on sentences to extract factual information. Syntactic transformations are based on English syntactic Rules.

  Below we discuss some theory related to Semantic Entailment which is necessary for extraction from facts from complex sentences

General Algorithm for Fact Extraction:

- Write rules for different grammatical Patterns which can be used to extract some facts.
- If Sentence contains these grammatical Patterns, Perform the logic for extraction of sentence by NLP transformations. Tregex (regular expression language for trees) is used for purpose of detection of patterns as well as rules for transformation and extraction.
- Original Sentence information is preserved and new extracted sentences are considered child of the this original sentence

6.2.1 Examples

Consider sentence: Tiger, Leopard and Moroccan Otter can be found in Taipei Zoo.

Following are extracted facts form the Noun Phrase Conjunction:

1. Tiger can be found in Taipei Zoo.
2. Leopard can be found in Taipei Zoo.
3. Moroccan Otter can be found in Taipei Zoo.

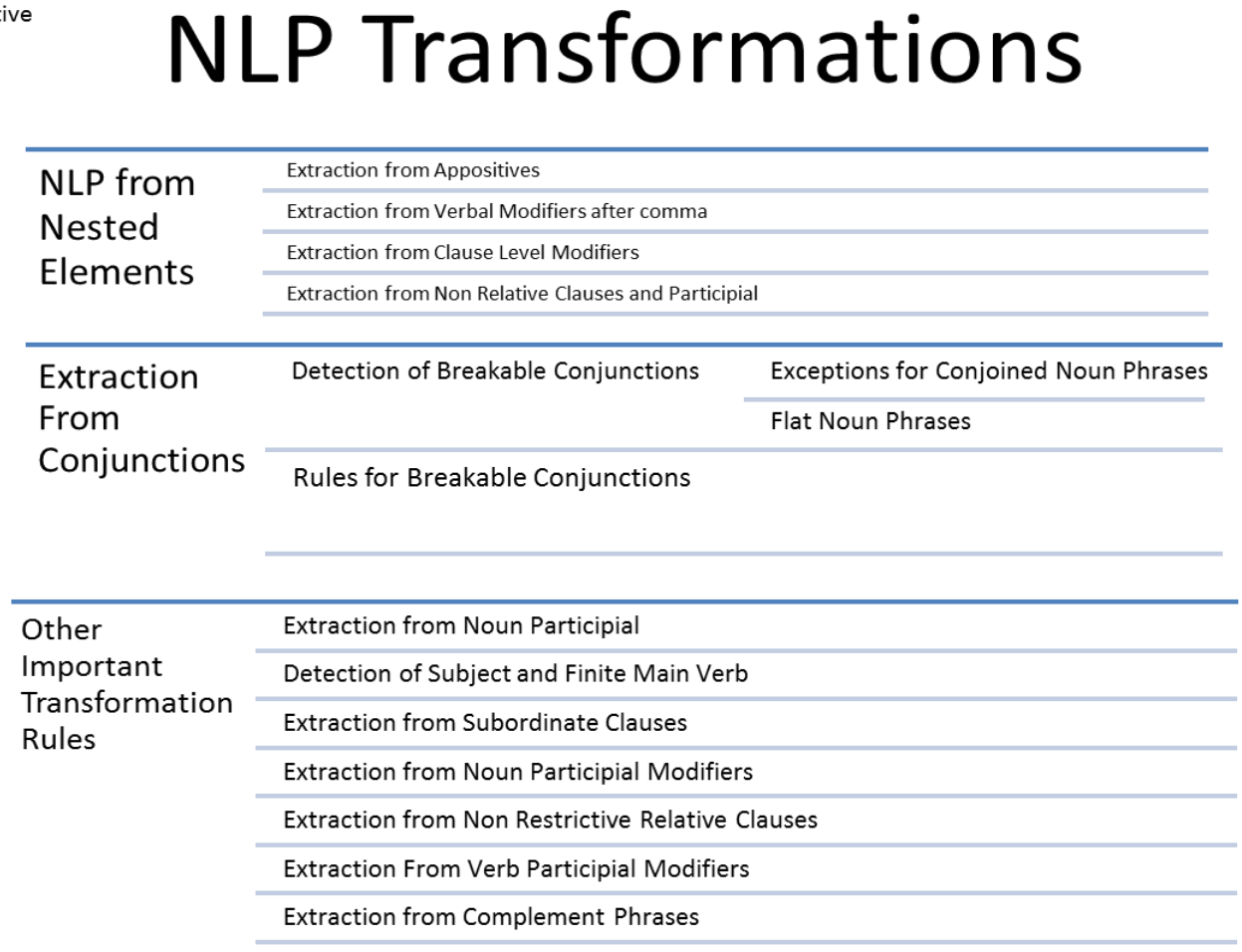
Consider example: John told me that he likes dogs but that he does not like cats.

Extracted Facts from the conjoined sbars:

1. John told me that he likes dogs.
2. John told me that he does not like cats.

Figure 4: Fact Extraction Module

Note: All these rules and their respective logic is already implemented



6.2.2 Extraction and Simplification by Semantic Entailment

Semantic entailment is defined as follows: A semantically entails B if and only if for every situation in which A is true, B is also true (Levinson, 1983, Chapter 4). We can extract simplifications from complex sentences by removing adjunct modifiers and discourse connectives and by splitting conjunctions of clauses and verb phrases. These transformations preserve the truth conditions of the original input sentence, while producing more concise sentences from which questions can be generated.

6.2.2.1 Removing Discourse Markers and Adjunct Modifiers

Many sentences can be simplified by removing certain adjunct modifiers from clauses, verb phrases, and noun phrases.

Consider example sentence:

1. *"**However**, Jefferson did not believe the Embargo Act, **which restricted trade with Europe**, would hurt the American economy."*

For example above, we can extract example by removing the discourse marker "**however** and the **relative clause** "which restricted trade with Europe".

2. *Jefferson did not believe the Embargo Act would hurt the American economy.*

Sentence 2 is true in all situations where sentence 1 is true, and is therefore semantically entailed.

Following properties can be observed about markers:

- Discourse markers such as "however" do not affect truth conditions in and of themselves, but rather serve to inform a reader about how the current sentence relates to the preceding discourse.

- Adjunct modifiers do convey meaning, but again do not affect semantic entailment. Of course, many adjuncts provide useful information for example, prepositional phrases that identify the locations and times at which events occurred can be the target of where and when questions, respectively.

Below are the types of adjuncts and discourse markers:

6.2.2.1.1 Different adjuncts and discourse markers

- **non-restrictive appositives** (e.g., Jefferson, the third U.S. President, . . .)

Appositives and relative clauses can be restrictive or non-restrictive. Restrictive versions resolve referential

ambiguities and are not typically offset by commas (e.g., the man who was tall . . .). The non-restrictive ones provide additional information and are offset by commas (e.g., John, who was tall.). Our implementation only extracts non-restrictive forms, and distinguishes them from restrictive forms by including commas in the Tregex tree searching rules.

- **non-restrictive relative clauses** (e.g., Jefferson, who was the third U.S. President, . . .)
- **parentheticals** (e.g., Jefferson (1743–1826) . . .)
- **participial modifiers of noun phrases** (e.g., Jefferson, being the third U.S. President, . . .)
- **verb phrase modifiers offset by commas** (e.g., Jefferson studied many subjects, like the artist Da Vinci)

Modifiers that precede the subject (e.g., Being the third U.S. President, Jefferson . . .)

6.2.2.2 Splitting Conjunctions

In most cases, the conjuncts in these con-junctions are entailed by the original sentence. For example, given John studied on Monday but went to the park on Tuesday, both John studied on Monday and John went to the park on Tuesday are entailed.

6.2.2.3 Extraction by Presupposition

In addition to the strict notion of semantic entailment, the pragmatic phenomenon of presupposition plays an important role in conveying information.

The semantically entailed information in complex sentences often covers only a fraction of what readers understand. Consider again the example about the Embargo Act:

Below are the categories for extraction by presupposition

- **non-restrictive appositives** (e.g., Jefferson, the third U.S. President, . . .)
- **non-restrictive relative clauses** (e.g., Jefferson, who was the third U.S. President, . . .)
- **participial modifiers** (e.g., Jefferson, being the third U.S. President, . . .)
- **temporal subordinate clauses** (e.g., Before Jefferson was the third U.S. President . . .)

6.2.3 Syntactic Rules for Sentence Complexity Reduction

6.2.3.1 Transform Quotations

6.2.3.1.1 Detect Quotations

```
ROOT < (S|SINV=mainclause < (NP|SBAR=subj !$++ /,/) < VP=mainvp  [ < (PP=modifier < NP) | < (S=modifier < SBAR|NP <<# VB|VBD|VBP|VBZ) ] )
```

6.2.4 Transform Leading Prepositional Phrases

6.2.4.1 Examples:

In January snow fell.
Snow fell in January.

On Tuesday, John ran.
John ran on Tuesday.

Because of the nice weather , John ran.
John ran because of the nice weather.

6.2.4.2 Detect Leading Prepostional Phrases

```
ROOT < (S|SINV=mainclause < (NP|SBAR=subj !$++ /,/) < VP=mainvp  [ < (PP=modifier < NP) | < (S=modifier < SBAR|NP <<# VB|VBD|VBP|VBZ) ] )
```

6.2.5 Extraction From Noun Participials

6.2.5.1 Examples

This was the book, written thousands of years ago.

The book was written thousands of years ago.

It was a blue car, bought by John.
The blue car was bought by John.

6.2.5.2 Identification of Noun Participials

```
ROOT < (S  [ << (NP < (NP=subj  $++ (/,/ $+ (VP=modifier <# VBN|VBG|VP=tense ))))  | < (S !< NP|SBAR < (VP=modifier <# VBN|VBG|VP=tense) $+ (/,/ $+ NP=subj))  | < (SBAR < (S !< NP|SBAR < (VP=modifier <# VBN|VBG=tense)) $+ (/,/ $+ NP=subj))  | < (PP=modifier !< NP <# VBG=tense $+ (/,/ $+ NP=subj)) ] )  <<# /^VB.*$/=maintense
```

6.2.6 Extraction From Nested Elements

6.2.6.1 Extraction from appositivies

6.2.6.1.1 Example

John, the painter, knew Susan.

6.2.6.1.2 Rules

```
NP=parent < (NP=child $++ (/,/ $++ NP|PP=appositive) !$-- /,/) !< CC|CONJP
```

6.2.6.2 Extraction From Verbal Modifiers after Comma

6.2.6.2.1 Example

John studied, hoping to get a good grade. -> John studied.

6.2.6.2.2 Detection Rule

```
ROOT=root << (VP! < VP < (/,/=comma $+ /[^`].*/=modifier))
```

6.2.6.3 Clause Level Modifiers

6.2.6.3.1 Example

However, John did not study. -> John did not study.

6.2.6.3.2 Detection Rule

```
ROOT=root < (S=mainclause < (/SBAR|ADVP|ADJP|CC|PP|S|NP/=fronted !< (IN <
if|unless) !$ ``  $++ NP=subject))
```

Non Restrictive Relative Clauses and Participials

6.2.6.3.3 Example

John, who hoped to get a good grade, studied. -> John studied.

6.2.6.3.4 Detection Rule

```
NP < (VP|SBAR=mod $- /,/=punc !$+ /,/ !$ CC|CONJP)
```

6.2.6.4 Parenthicals

6.2.6.4.1 Example

John Smith (1931-1992) was a fireman. -> John Smith was a Fireman.

6.2.6.4.2 Detection Rule

```
__=parenthetical [ $- /-LRB-/=leadingpunc $+ /-RRB-/=trailingpunc  | $+
/,/=leadingpunc $- /,/=trailingpunc !$ CC|CONJP  | $+ (/:/=leadingpunc < --) $-
(/:/=trailingpunc < /--/) ]
```

6.2.6.5 Extraction from Conjunctions

6.2.6.5.1 Detection Of Breakable Conjunctions

6.2.6.5.1.1 Exceptions for Conjoined Noun Phrases

Example

John and Mary are two of my best friends.

Rule

```
NP=parent < (CONJP|CC !< or|nor [  $+ /^(N.*|PRP|SBAR)$/=child $-- /^(N.*|PRP|SBAR)$/
|  $-- /^(N.*|PRP|SBAR)$/=child $+ /^(N.*|PRP|SBAR)$/ ] )  !>> (/.*/ $ (CC|CONJP !<
or|nor))  !$ (CC|CONJP !< or|nor) !.. (CC|CONJP !< or|nor > NP|PP|S|SBAR|VP) !>> SBAR
$+ (ADVP < RB $+ VP)  | $- (/VB.*/ >> (VP $- (NP < CD)))  | >> (PP $- (NP < CD))
```

6.2.6.5.1.2 Rules for Breakable Conjunctions

Conjoined VPs, Clauses etc

```
CONJP|CC !< either|or|neither|nor > S|SBAR|VP [ $ SBAR|S | !>> SBAR ]
```

6.2.6.5.2 Clauses Conjoined by Semi Colons

```
S < (S=child $ (/:/ < /;/) !$++ (/:/ < /;/) )
```

6.2.6.5.2.1 Flat NPs

```
CONJP|CC !< either|or|neither|nor > NP !>> SBAR  !> (NP < (/^(N.*|SBAR|PRP)$/ !$
/^(N.*|SBAR|PRP)$/))
```

6.2.6.6 Detection of Subject and Finite Main Verb

```
ROOT  <+(S) NP|SBAR  <+(VP|S) VB|VBD|VBP|VBZ  !<+(VP) TO
```

6.2.6.7 Extraction from SubOrdinate Clauses

6.2.6.7.1 Example

As John slept, I studied. -> John slept.

6.2.6.7.2 Rules

```
SBAR [ > VP < IN | > S|SINV ]  !< (IN < if|unless|that) < (S=sub !< (VP < VBG))
>S|SINV|VP
```

6.2.6.8 Extraction From Noun Participial Modifiers

6.2.6.8.1 Example

6.2.6.8.2 Rules

```
ROOT < (S  [ << (NP < (NP=subj  $++ (/,/ $+ (VP=modifier <# VBN|VBG|VP=tense ))))  | <
(S !< NP|SBAR < (VP=modifier <# VBN|VBG|VP=tense) $+ (/,/ $+ NP=subj))  | < (SBAR <
(S !< NP|SBAR < (VP=modifier <# VBN|VBG=tense)) $+ (/,/ $+ NP=subj))  | <
(PP=modifier !< NP
```

6.3 Module 3: Entity Extraction Module

- The Entity extraction module extracts the entities from the text.
- The entity extraction uses following information :
  - Tagged Super Sense Information for sentence
  - Parse Trees
  - Ability to Detect Head Noun in an Noun Phrase
- The extracted noun entities are added in Senses. Following are the current sense of entities:

Figure 5: Wordnet Sense Information

Verb Senses	Noun Senses
•verb.body •verb.change •verb.cognition •verb.communication •verb.competition •verb.consumption •verb.contact •verb.creation •verb.emotion •verb.motion •verb.perception •verb.possession •verb.social •verb.stative •verb.weather	•noun.Tops •noun.act •noun.animal •noun.artifact •noun.attribute •noun.body •noun.cognition •noun.communication •noun.event •noun.feeling •noun.food •noun.group •noun.location •noun.motive •noun.object •noun.other •noun.person •noun.phenomenon •noun.plant •noun.possession •noun.process •noun.quantity •noun.relation •noun.shape •noun.state •noun.substance •noun.time

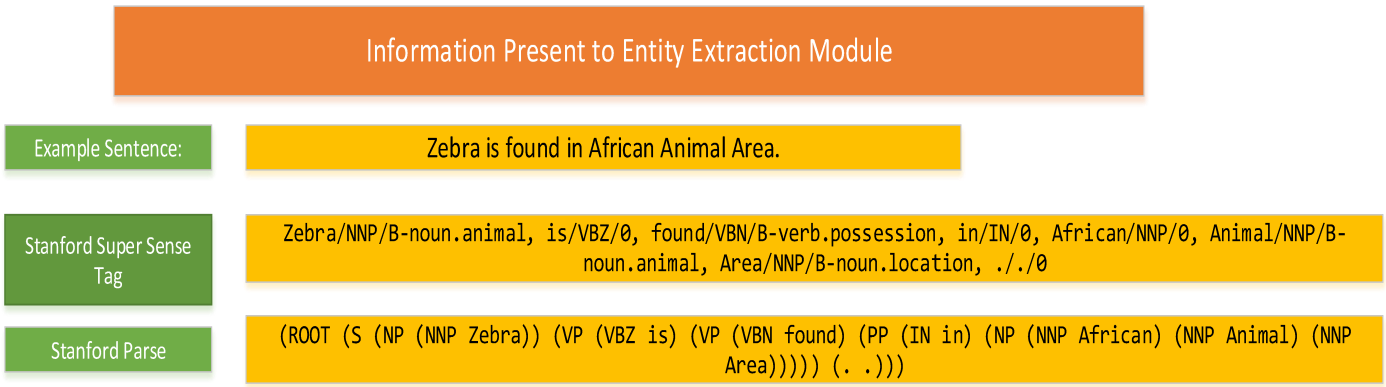
- Please note that these sense classes help to extract fairly accurate Entities including complex entities. Their types maybe wrong but entity phrases extraction is quite accurate.
  - E.g. African Animal Area is an entity. It is possible that it's tagged entity type: Artifact is not accurate. However the entity extraction is correct:
    - Entity is "African Animal Area" and not African, Animal, Area, African Animal, Animal Area etc.
- Adjectival Modifiers are extracted and stored as modifiers of the entities.
  - Big dog. Here the entity is : Dog (Type: Animal) and its modifier is Big (adjective)
  - Entity: African Animal Area (Here the whole phrase is an entity as each word is a noun)

6.3.1 Old Entity Extraction Module

6.3.1.1 Algorithm with Example

Let us consider an example to illustrate the Extraction process

Figure 6: Example of Sentence Information available to Entity Extractor



Algorithm for the current Entity Extraction Module is:

1. For each word, extract the Head noun of the word.
   - ✓ E.g. African, Head Noun: Area, Zebra, Head Noun: Zebra
2. Using the Stanford Parse tree, get the Parent noun phrase of the extracted head noun.
   - ✓ `(NP (NNP African) (NNP Animal) (NNP Area))) for African`
   - ✓ `(NP (NNP Zebra)) for Zebra.`
3. `Get the Sense Tag of the head noun. This will be the sense of the whole entity phrase. E.g. Sense tag of Area is Location, hence Sense of "African Animal Area" is Location`
4. If the Entity contains modifiers, in the form of adjectives, Extract them and add them as modifiers. E.g. In case of entity phrase: Big dog: big is modifier.
5. Add the extracted entities to a global data structure. This global looks like :
   - ✓ Map<NERNounTags, Map<Concept, List<Integer>>> Entity Tags
     - ✓ Here NERNounTags are the Sense of Nouns mentioned in here: on page 32

6.3.1.2 Key Points Of Algorithm

- The above algorithm is performed for each Sentence, and the added entity is updated in the Global data structure.
- If the entity already exists in the Data structure, the sentence number is added to existing data structure for reference information.
  - This way, We can know what entities are present , in what sentence numbers
  - The same approach is used, whether the sentence is a dialogue, article etc.

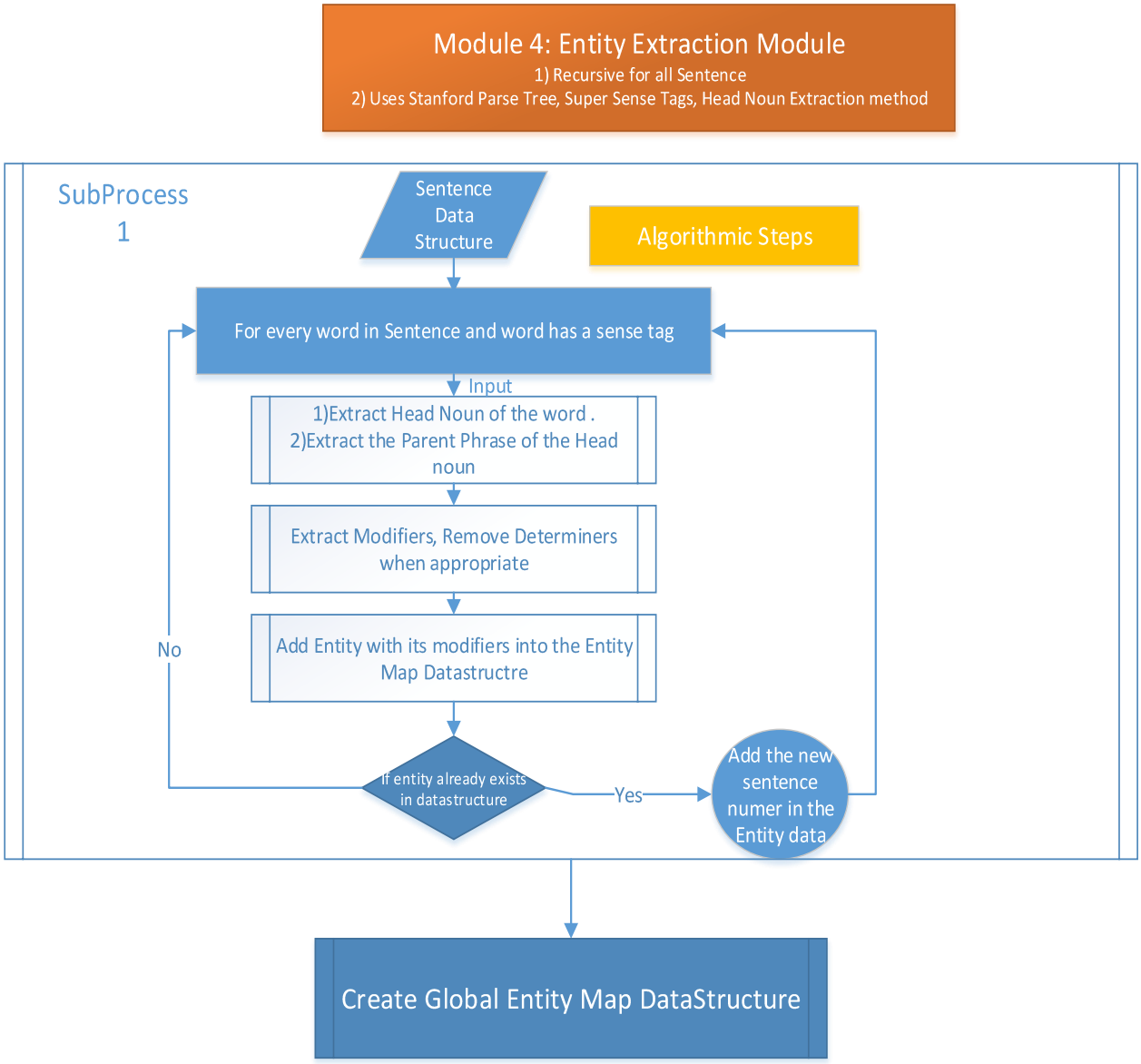
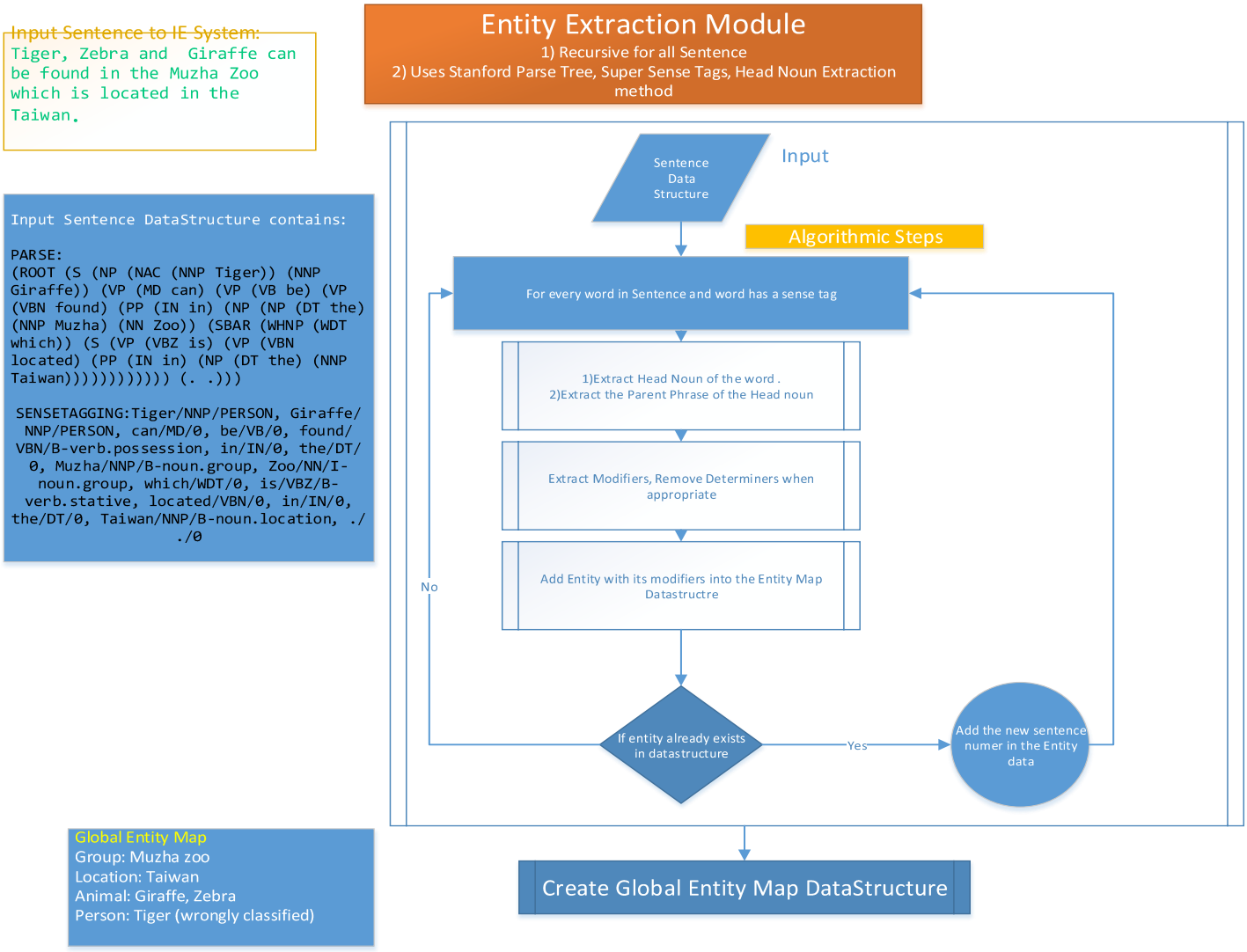


Figure 7: Module 4: Entity Extraction Module

6.3.1.3 Example Flow

Figure 8: Module 4: Entity Extraction Example Flow



6.3.2 New Entity Extraction Module

In the New entity extraction module, addition will be the usage of the Module 5: Ontology Information.

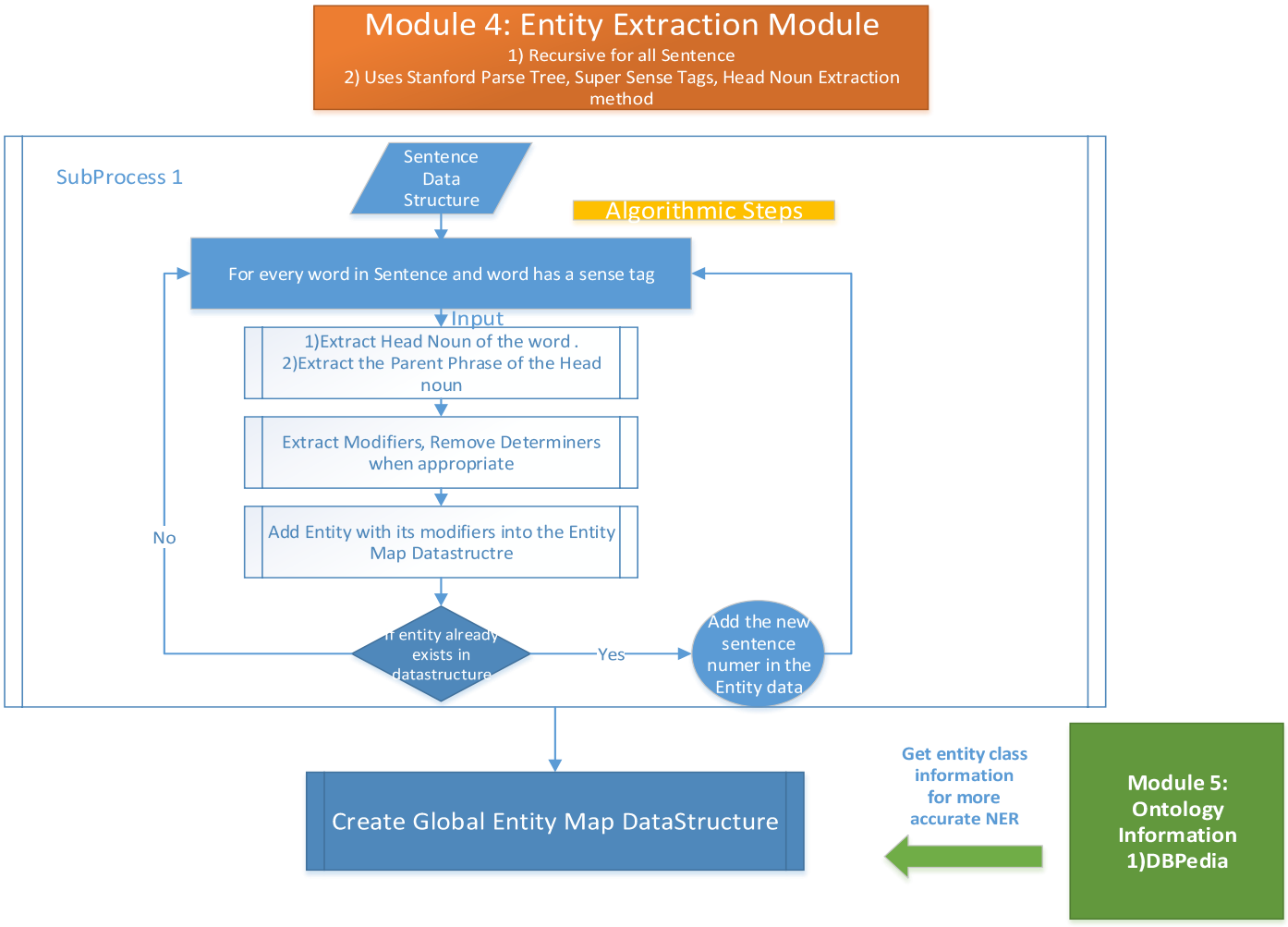
Key Points:

- The more accurate entity types will be obtained using domain ontologies, bigger general ontologies like DBPedia.
- The entity phrase extraction is not dependent on domain. However, the entity types certainly are.
- This also means we have entity types classes dynamically created from the types returned by domain ontologies. E.g. Device is an entity type for “Seagate Hard disk”. This entity type is obtained from Computer Domain ontology, also General DBPedia ontology.

6.3.2.1 Things to Accomplish

- All Domain ontologies, Application Ontologies will be defined in uniform way. The entity information should be queried in uniform way, whether we want entity information from application or domain ontologies.
- SPARQL Queries to get entity information from the Ontologies.
  - Example: DBPedia has a SPARQL query endpoint to retrieve information.
    - Ganesh's NER extraction is based on writing queries to get entity type and other information from DBPedia endpoint.

Figure 9: Module 4: New Entity Extraction Module



6.4 Module 4: Relation Extraction Module

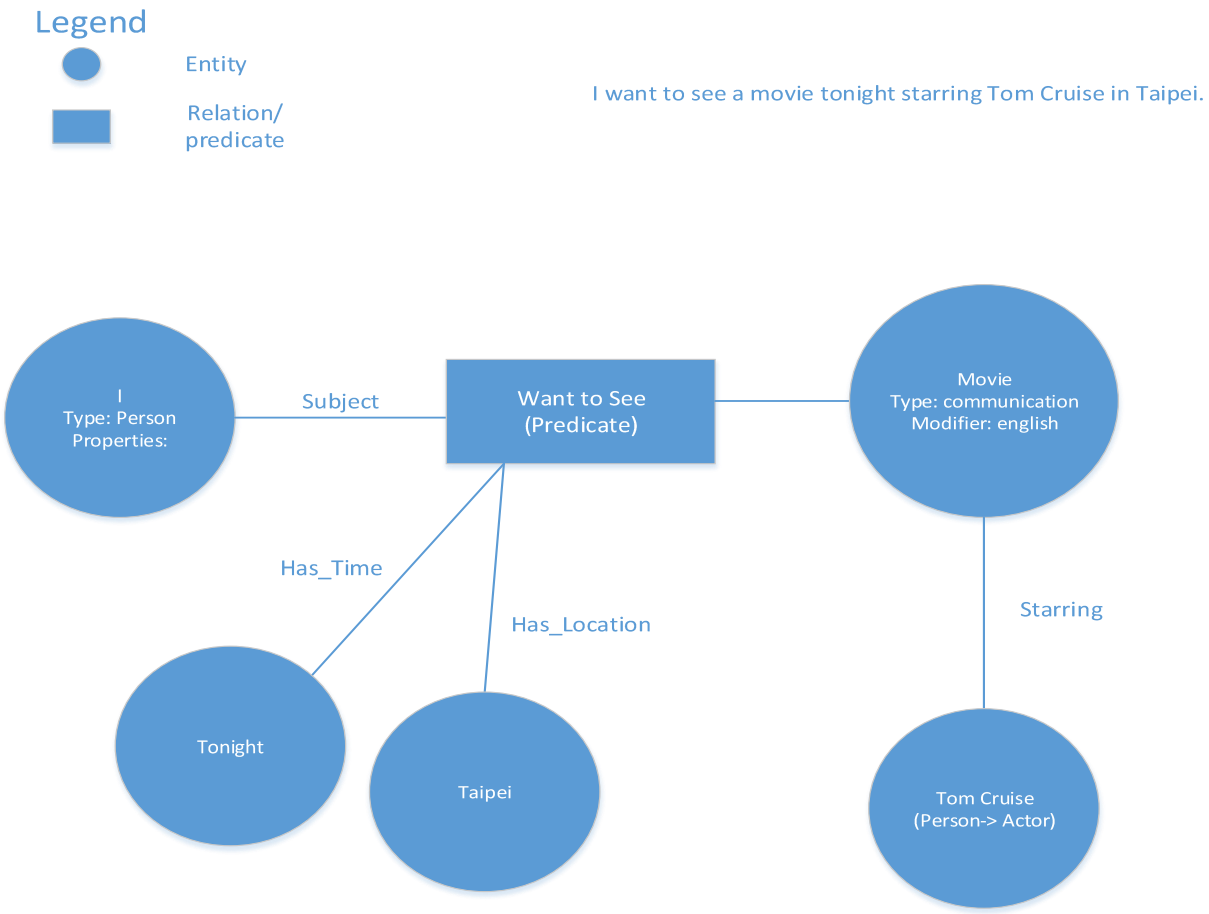
Relation Extraction Module will extract the triplet: predicate, subject, object which will be present in

sentences. For complex sentences, more than one triplet can be present.

The objective would be to extract relations which would help create the following graph:

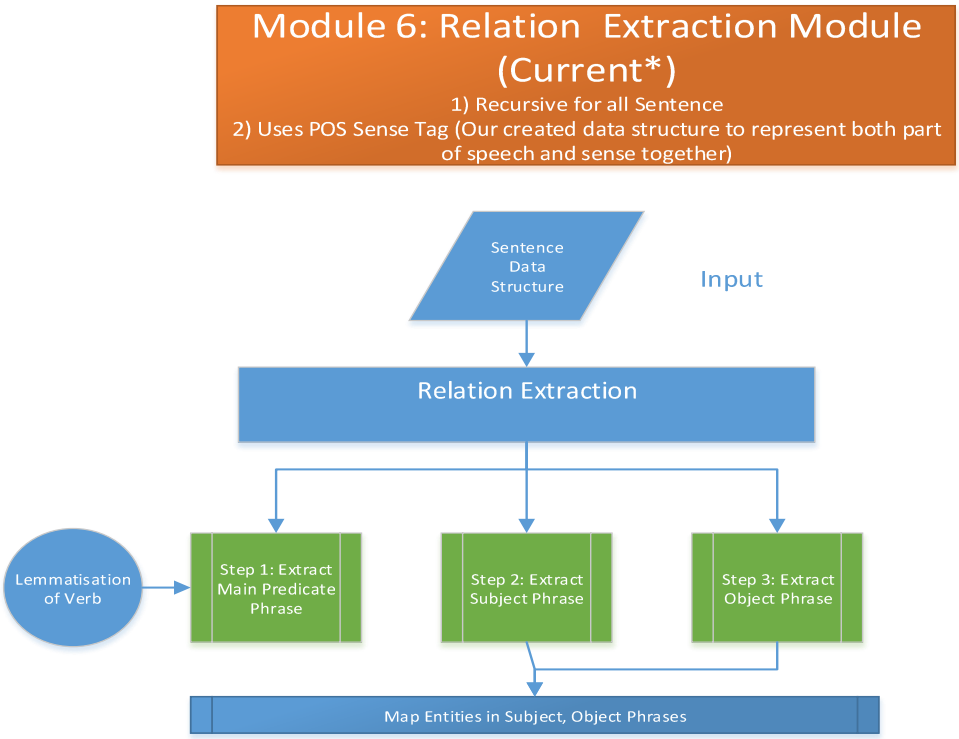
Example Sentence:
Sentence: I want to see a movie tonight starring Tom cruise in Taipei.

Figure 10: Example of Knowledge Graph used for Relation Extraction Module



6.4.1 Current Relation Extraction Module

Figure 11: Relation Extraction Module (Current)



6.4.1.1 Algorithm with Example

- Using Sense Tag information, identify the main entry verb of the sentence. In this case, wants is the main entry verb indicated by B-verb.emotion.
- Traverse to extract the predicate phrase. The predicate phrase can include determiners, prepositions, common nouns but not the entities
- Extract the subject phrase and the object phrase. The boundaries are generated during the predicate phrase extraction
- Link the Entities present in the sentence to subject, object phrase

Figure 12: Example flow of current Relation Extraction Module

Input Sentence: John wants to go for scuba diving tomorrow afternoon .

Sentence Data Structure contains:

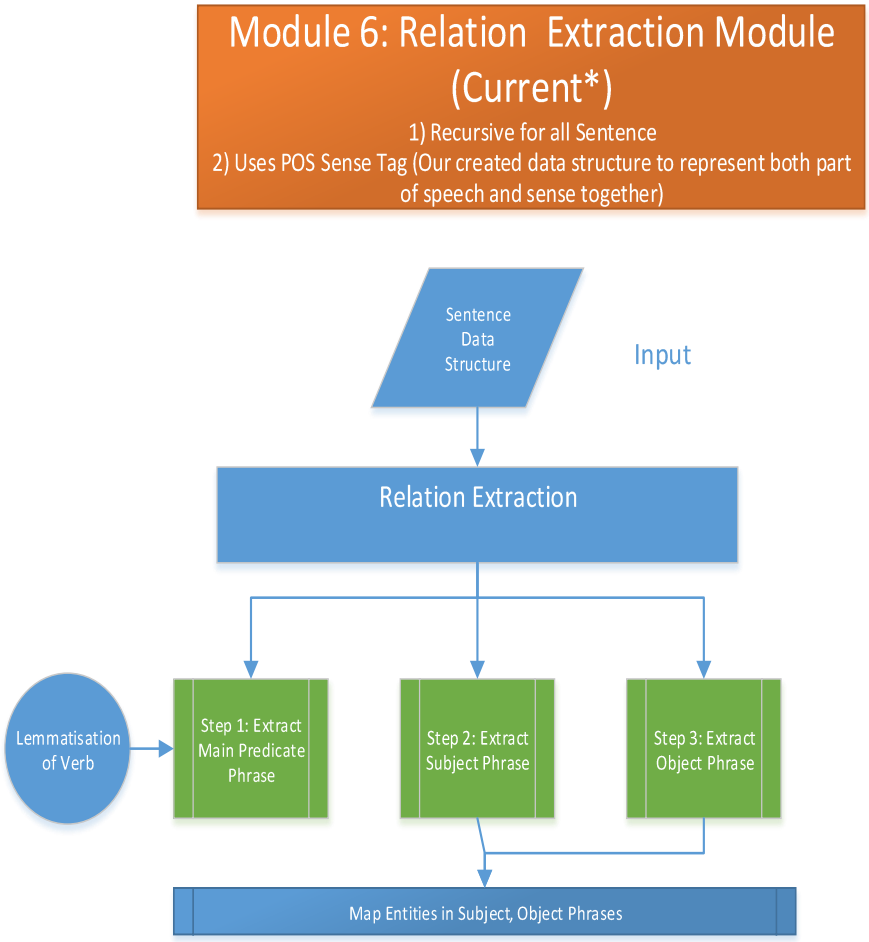
Parse: (ROOT (S (NP (NNP John)) (VP (VBZ wants) (S (VP (TO to) (VP (VB go) (PP (IN for) (NP (NP (NN scuba)) (NP (JJ diving) (NN tomorrow)))) (NP (NN afternoon)))))) (. .)))

Entities:
Person: John
Act: Scuba Diving
Time: Tomorrow

SENSE :John/NNP/PERSON, wants/VBZ/B-verb.emotion, to/TO/0, go/VB/0, for/IN/0, scuba/NN/B-noun.act, diving/JJ/I-noun.act, tomorrow/NN/B-noun.time, afternoon/NN/B-noun.time, ././0

SUBJECT: john
PREDICATE: want to go for
OBJECT: scuba diving tomorrow afternoon

John(person), scuba diving (act), tomorrow afternoon (time) are mapped entities



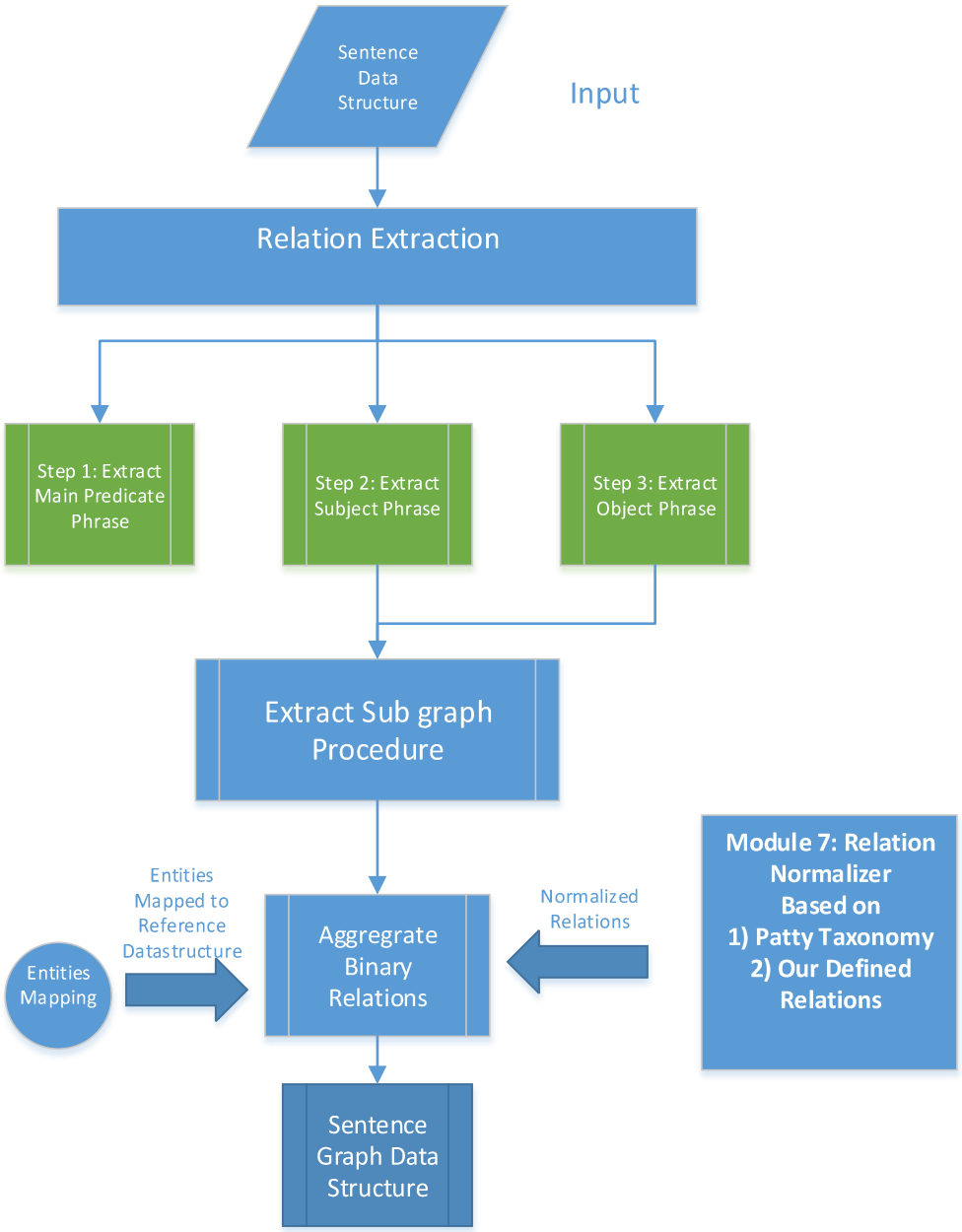
6.4.2 New Relation Extraction Module

- In the New Relation Extraction Module we want to extract from the Compound Subject and Object Phrase.
- The set of relational data should be sufficient enough to create the knowledge graph data structure.
- Relation Normalization has to be performed to have a set of normalized relations on which we can perform inference.

Figure 13: New Relation Extraction Module

Module 6: Relation Extraction Module (New*)

1) Recursive for all Sentence
2) Uses POS Sense Tag (Our created data structure to represent both part of speech and sense together), Dependency Parse



6.5 Module 8: Knowledge Graph Creator

- This module will generate the graph data structure.
- The generated graph can be visualized graphically
- API can provide information from the graph data structure.

6.6 Module 9: Text Knowledge Aggregrator

- In Case the Text is Article. It should contain lot of information

6.7 Module 3: Paragraph Level Discourse

This module can detect paragraph level discourse information. The paragraph level discourse information is useful when dealing with text

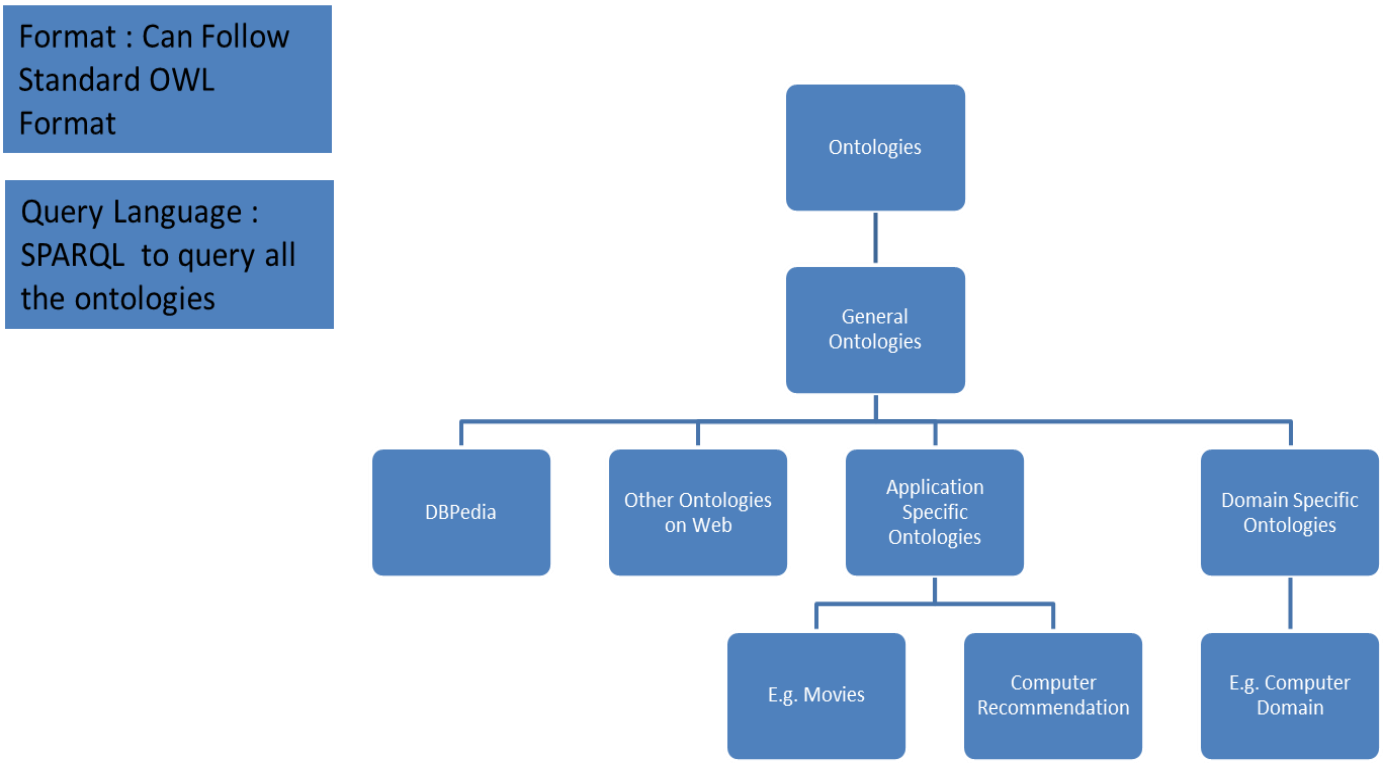
6.8 Module 5: Ontology Information

This module provides information on entities from the ontologies.

Ontologies can be:

- Domain Ontologies
- Application Specific Ontologies
- General Ontologies
-

Figure 14: Module 5: Ontology Information



6.8.1 Uniform Ontologies

All the ontologies will be in the same format.

6.8.2 Uniform and singular way of accessing information from ontologies

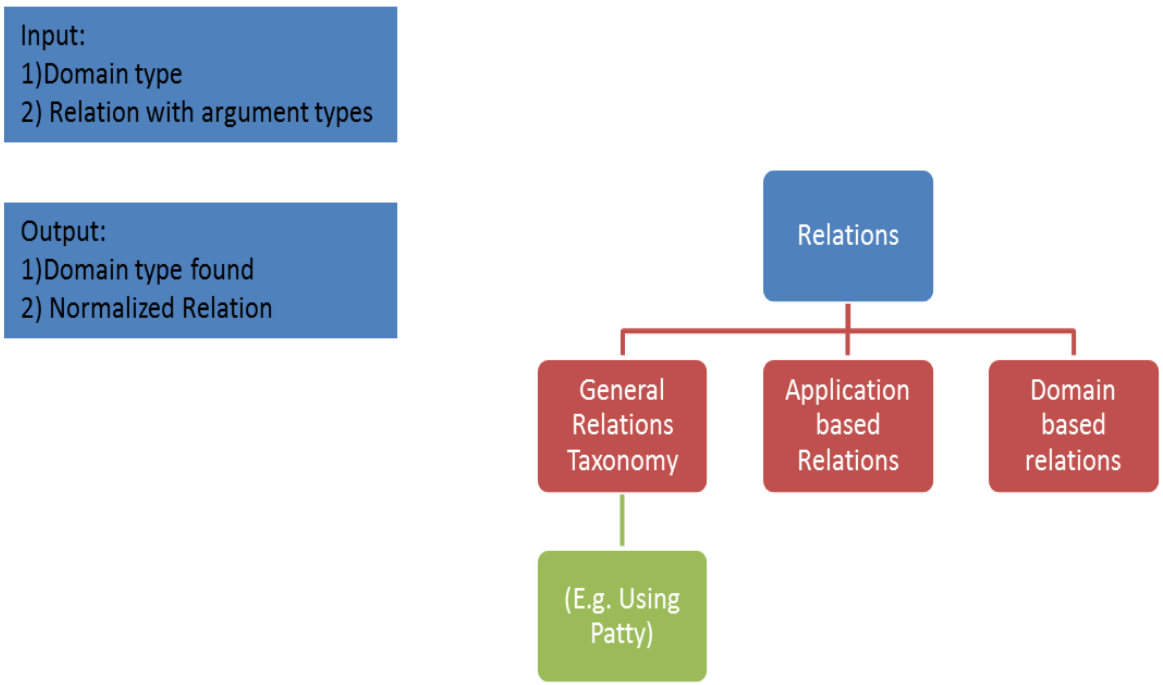
Information from the Ontologies can be accessed in Uniform manner.

All the ontologies can be queried for information using SPARQL language.

6.9 Module 7: Relation Normalizer

- The Relation Normalization module can provide the normalized relation for the extracted relation.
- There is another option where we only extract the defined set of relations. This defined set can be obtained from some research group.

Figure 15: Relation Normalization Module



6.10 Module 10: Summarization Module

This module can provide summarization information when the input is some article.

7 Policies and Tactics

7.1.1 Tactics

- This project is a team project with Collaborative Effort.
- Team Project Management policies will be followed.

- Each Module of the System should be developed independently.

8 Glossary

To be added

9 Bibliography

DBPedia: : http://dbpedia.org/

- Entity Extraction: From Unstructured Text to DBPedia RDF Triples

- Sumo: http://www.ontologyportal.org/
  - A comparison of Upper Ontologies
    - http://www.disi.unige.it/person/MascardiV/Download/DISI-TR-06-21.pdf
- Patty taxonomy:
  - A taxonomy of Relational Patterns with Semantic Types
    - http://www.mpi-inf.mpg.de/~nnakasho/papers/patty-emnlp12.pdf