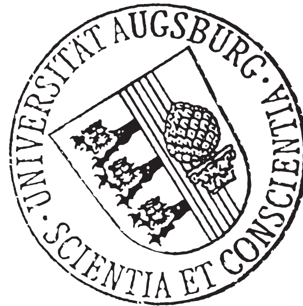
Linguistically Motivated Ontology-Based Information Retrieval

Dissertation

for the degree of

Doctor of Natural Sciences (Dr. rer. nat.)



Wolf Fischer

University of Augsburg

Department of Computer Science

Software Methodologies For Distributed Systems

February 2013

Linguistically Motivated Ontology-Based Information Retrieval

Supervisor: **Prof. Dr. Bernhard Bauer**, Department of Computer Science, University of Augsburg, Germany

Advisor: **Prof. Dr. Elisabeth André**, Department of Computer Science, University of Augsburg, Germany

Date of defense: April 22nd, 2013

Copyright © Wolf Fischer, Augsburg, February 2013

Abstract

When Tim Berners-Lee proposed his vision of the Semantic Web in 2001, he thought of machines that automatically execute specific tasks based on available knowledge. The knowledge should be captured within ontologies which provide an unambiguous and semantically rich way to capture information. The information could further be used to enhance tasks like information retrieval, i.e., the retrieval of documents which match specific criteria.

Over a decade later, technologies which are required for the Semantic Web have been established in several areas, e.g., the biological and medical domains. Both share a very constant pool of knowledge, which does not change as rapidly as in other domains, i.e., neither a lot of new knowledge must be added continuously nor the existing knowledge has to be updated very often. These circumstances make both domains suitable for manually creating ontologies. However, in case of a domain with constantly incoming new knowledge, it would be a great advantage if this knowledge could automatically be added or matched to an ontology. However, there is nearly no concept available on how ontological knowledge can be mapped to natural language precisely.

We therefore developed the *SE-DSNL* approach. It provides experts with the ability to specify how ontological knowledge can be mapped to linguistic information of any known language. The concept provides a flexible and generic meta model which captures all the relevant information. In order to use this for parsing natural language text a prototypical implementation has been developed which takes the information of a SE-DSNL model and applies it to a given input text. The result is a semantic interpretation of the input text which maps its lexical and syntactic elements to the ontology. The direct integration of semantic and linguistic information further allows using the semantic information at runtime. This yields certain advantages which are demonstrated by treating elaborate linguistic phenomena like pronominal anaphora resolution, word sense disambiguation, vagueness and reference transfer. To show the validity of the approach it has been evaluated using scenarios and two case studies.

Zusammenfassung

Als Tim Berners-Lee seine Vision des 'Semantic Web' 2001 vorstellte, dachte er an Maschinen, die, basierend auf vorhandenem Wissen, automatisch verschiedene Aufgaben erledigen können. Das Wissen sollte in Form von Ontologien vorliegen, welche einen eindeutigen und semantisch mächtigen Weg darstellen, Informationen zu erfassen. Weiterhin könnten diese Informationen für Anwendungen wie die Informationsrückgewinnung verwendet werden, d.h. das Auffinden von Dokumenten, die gewissen Kriterien entsprechen.

Über ein Jahrzehnt später zeigen die Technologien des Semantic Web insbesondere in biologischen und medizinischen Bereichen ihre Vorteile. Beide Bereiche eint ein etabliertes Kernwissen, welches nur selten angepasst werden muss. Dadurch sind diese Bereiche geeignet, die für sie notwendigen Ontologien manuell zu erstellen. In anderen Bereichen (d.h. Bereiche mit sich häufig veränderndem Wissen) wäre es jedoch von Vorteil, wenn neue oder geänderte Informationen automatisch in die Ontologie übertragen werden könnten. Das Problem jedoch ist, dass es kaum Ansätze dafür gibt, wie ontologisches Wissen mit natürlicher Sprache zusammengebracht werden kann.

Zu diesen Zweck wurde der *SE-DSNL* Ansatz entwickelt. Er ermöglicht es Experten, die Verbindungen zwischen ontologischem und linguistischem Wissen von beliebigen Sprachen zu definieren. Im Zentrum des Ansatzes steht ein flexibles und generisches Metamodell, welches alle notwendigen Informationen erfassen kann. Eine prototypische Implementierung analysiert auf Basis eines SE-DSNL Modells einen natürlich sprachlichen Text. Das Ergebnis ist eine semantische Interpretation, welche die Verbindungen zwischen dem Text und der Ontologie sowohl auf lexikalischer als auch syntaktischer Ebene enthält. Zusätzlich ermöglicht diese direkte Integration der semantischen und linguistischen Informationen die Verwendung des ontologischen Wissens zur Laufzeit. Die dadurch entstehenden Vorteile werden durch die Behandlung von komplizierten linguistischen Phänomenen wie pronominaler Anapher Auflösung, Wort-Sinn Disambiguierung, Unbestimmtheit und Referenz Transfer demonstriert. Der Ansatz wird anhand von Szenarien und zwei Fallstudien evaluiert.

Acknowledgements

On long journeys you sometimes have the honor and pleasure of meeting different persons who help and guide you on your way. This is the place, where I want to express my gratitude to all the people that accompanied and helped me in creating this thesis.

First and foremost, I especially want to thank Prof. Dr. Bernhard Bauer for giving me the opportunity to become a member of his team and graduate under his supervision. His kind support, motivation and guidance always provided me with the freedom and energy to both create and develop my own ideas.

I thank Prof. Dr. Elisabeth André, who accepted to be advisor of my thesis.

Working in an environment like the one which my current and prior colleagues created can not be taken for granted. Discussions both about private and research related topics as well as continuous support in all different kinds of matters made these past couple of years a cheerful, productive and balanced working experience. I am very grateful for having been part of this team.

Over the years, several students were actively involved in my research. Some of them were assistants to my research over longer periods of time and / or wrote their Bachelor or Master thesis under my guidance. All of them helped me in creating this thesis by bringing in ideas of their own, implementing applications or discussing specific aspects with me. I am thankful to all of them.

I am very grateful to all my friends and family, which always motivated and helped me when I needed it. I especially want to thank my parents Angelika and Robert which provided me with a good education and always supported me in giving my best.

Last but not least I especially want to thank Sylvia, which experienced these last couple of years like no other person. Her constant believe in me and my thesis greatly encouraged and helped me finishing this thesis.

Contents

Acknowledgements vii

1. Introduction 1

1.1. Introduction and Motivation 2
1.2. Problem Definition 5
1.2.1. Ontologies and Natural Language 5
1.2.2. Pipelined processing 6
1.2.3. Alignment of Ontology and Natural Language Text 7
1.2.4. Ambiguity, Vagueness and Reference Transfer 9
1.2.5. Retrieving Information from Semantic Interpretations 9
1.3. Objectives 11
1.3.1. Combining Ontologies with Natural Language 11
1.3.2. Concurrent Analysis of Natural Language Specific Problems 11
1.3.3. Semantic Interpretation of Natural Language Text 12
1.3.4. Handling Ambiguities, Vagueness and Reference Transfer 12
1.3.5. Semantic Information Retrieval 13
1.4. Architecture 14
1.5. Publications 15
1.6. Outline 16

2. Basics 19

2.1. Natural Language 20
2.1.1. Morphology 20
2.1.2. Syntax 22
2.1.3. Meaning 26
2.1.4. Ambiguity, Vagueness and Reference Transfer 28
2.1.5. Cognitive Linguistics and Construction Grammar 29
2.1.6. Computational Linguistics 33
2.2. Ontology 40
2.2.1. RDF and RDFS 41
2.2.2. OWL 44

2.2.3. Linguistic grounding . . . 47
2.3. Ontology-based Information Retrieval & Extraction . . . 52

3. Combining Ontological and Linguistic Information **57**
3.1. Introduction . . . 58
3.2. Requirements . . . 60
3.3. SE-DSNL Meta Model . . . 63
3.3.1. Scope . . . 64
3.3.2. Semantic Scope . . . 65
3.3.3. Syntactic Scope . . . 67
3.3.4. Construction Scope . . . 67
3.3.5. Interpretation Scope . . . 68
3.4. Formal Specification . . . 71
3.5. Creating Knowledge . . . 77
3.5.1. OWL to SemanticScope Transformation . . . 77
3.5.2. Linguistic Representation of SemanticElements . . . 83
3.5.3. Mapping Syntactic and Semantic Structures . . . 85
3.5.4. Modeling Guidelines . . . 87
3.5.5. Multilingual Representation . . . 92
3.5.6. Conclusion . . . 93
3.6. Related Work . . . 94

4. Semantic Interpretation of Natural Language **97**
4.1. Introduction . . . 98
4.2. Semantic Interpretation . . . 99
4.2.1. Syntax Tree Alignment . . . 99
4.2.2. Construction Application . . . 104
4.2.3. Solution Extraction . . . 114
4.2.4. Creating the Interpretation Model . . . 120
4.2.5. Detecting Termination Cases . . . 121
4.2.6. Handling runtime complexity . . . 124
4.3. Functions . . . 125
4.3.1. Correct Word Order . . . 125
4.3.2. Pronominal Anaphora Resolution . . . 129
4.3.3. Check and Create Triple . . . 136
4.3.4. Semantic Type Check . . . 137
4.4. Related Work . . . 138
4.4.1. Ontology-based Information Retrieval . . . 138
4.4.2. Ontology-based Information Extraction . . . 141
4.4.3. Semantic Annotation . . . 143

4.4.4. Conclusion 143

5. Semantic Spreading Activation 147
5.1. Introduction 148
5.2. Requirements 150
5.3. Example 151
5.4. Definitions 152
5.5. Initialize tokens 155
5.6. Create the Token flow 157
5.7. Analyze the token flow 159
5.8. Termination 163
5.9. Result Computation 165
5.10. Related Work 170

6. Semantic Information Retrieval and Classification 173
6.1. Introduction and Motivation 174
6.1.1. Problem Definition 175
6.1.2. Requirements 176
6.2. Pattern Metamodel 178
6.3. Pattern Semantics 181
6.3.1. Target Cardinality 183
6.3.2. Necessity 183
6.3.3. Extended Matching 184
6.3.4. Transitivity 185
6.4. Pattern Evaluation 188
6.4.1. Phase 1 - Selection 188
6.4.2. Phase 2 - Evaluation 191
6.5. Related Work 197

7. Evaluation 199
7.1. Introduction and Motivation 200
7.2. Framework Architecture 202
7.3. Scenario-based Evaluation 205
7.3.1. Modifiability 205
7.3.2. Reusability 209
7.3.3. Performance 212
7.4. Case Study 1 214
7.4.1. Case Study Design 215
7.4.2. Results 223
7.4.3. Discussion 225

7.4.4. Conclusion . . . 230
7.5. Case Study 2 . . . 232
7.5.1. Case Study Design . . . 233
7.5.2. Results . . . 242
7.5.3. Discussion . . . 243
7.5.4. Conclusion . . . 248

8. Conclusion **249**
8.1. Summary . . . 250
8.2. Outlook . . . 253

A. Appendix **255**
A.1. POS Tags and Syntactic Categories from Penn Treebank . . . 256
A.2. Case Study 1 Texts . . . 257
A.3. Case Study 2 Commands . . . 258

Acronyms **259**

Bibliography **261**

List of Figures **279**

List of Tables **283**

List of Algorithms **285**

1

Introduction

1.1. Introduction and Motivation

For decades people have been dreaming of building machines which would help them in their every day lives. Such machines should not only be capable of doing 'ordinary' stuff like bringing, e.g., a cup of coffee from A to B, but *understand* information which is given to them. In 2001, Tim Berners-Lee et al. wrote an article, in which they coined the term 'Semantic Web' [1]. The article outlines how information that are meaningful not only to humans but also to machines could improve the life of humans on earth. The important prerequisite, however, is that there must be a language which is capable of capturing all the necessary information such that machines can understand what they have to do. The concept of *ontologies* has been described as a way to achieve the goal. An ontology is "a machine-processable specification with a formally defined meaning" [2] and allows to capture the knowledge of a specific domain, e.g., a company. With this data a machine can automatically infer new information. Such knowledge can, however, also be used to enhance the precision of *information retrieval*, basically the task which search engines perform (for details see section 2.3). This means that if websites or documents could be linked to ontologies, it would be possible to not only search for words, but instead look for specific meaning, which could greatly increase the overall performance of information retrieval. Also, if such ontologies could capture enough knowledge, it might be used by machines to automatically execute certain tasks.

Ontologies bring many new possibilities which can enhance the overall quality of life by either accelerating certain tasks or enabling machines to do them. The reason is that ontological knowledge is unambiguous and logical. These characteristics are a huge difference to what humans use for communication and what provides so many problems for automatic processing: Natural language. It allows persons to transfer information to other persons. The information can be anything, ranging from concrete situations they experienced to abstract mathematical facts they learned about. Natural language is the preferred way for people to communicate. However, when it comes to computers trying to understand natural language, there is a wide variety of complex problems they have to deal with. There are, e.g., things like ambiguity, i.e., words may have multiple meanings, sentences can be interpreted in different ways (and therefore also have multiple meanings), etc. There exist phenomena like metaphors, one of the most important ways of languages to create new words by describing situations with expressions which so far have only been used in other contexts [3]. All these different characteristics of natural language make it extremely difficult for computers to decipher its meaning.

In order to go a first step in the direction of machine understandable information in times of the internet, the World Wide Web Consortium (W3C) [4] created different standards [5], which can be seen in figure 1.1. The language which is meant to be understood

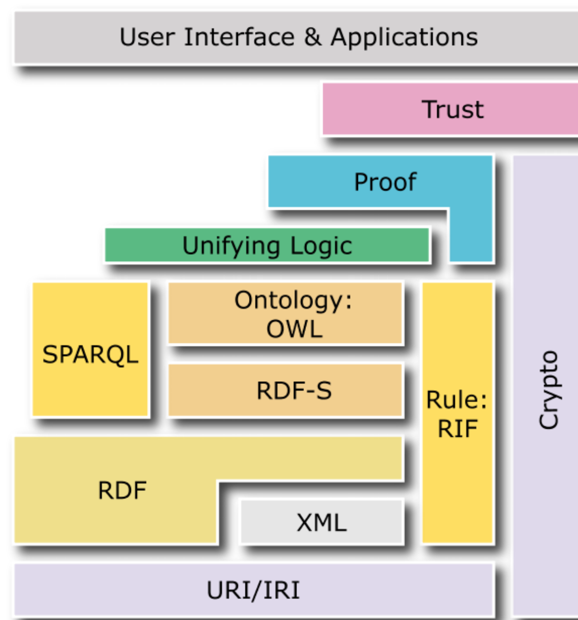


Figure 1.1.: The Semantic Web Stack according to the W3C [5]

by machines, is the Web Ontology Language (OWL). OWL allows the representation of ontological knowledge and can be used by computers to infer new knowledge or verify existing information. Different companies and researchers use OWL successfully in a variety of scenarios [6] [7] [8]. Some situations are especially well suited for ontologies, whereas others are much more difficult to handle. The problem is that in order to work with ontological knowledge it must be created first. Depending on the scenario at hand it might also be necessary to constantly update existing or create new knowledge. This is a challenge which quickly becomes very time consuming and complex. One approach to solve this problem has been to divide the knowledge acquisition problem into smaller pieces. Different people could then work on just one small part, thereby accelerating the process [9]. However, the past decade showed that the approach is difficult to maintain. Therefore, researchers started to develop concepts which try to automatically extract relevant information from natural language text. This discipline is called *information extraction*. Much like information retrieval, information extraction analyzes natural language text and tries to identify certain patterns which could represent relevant information. This is extracted and stored separately. Ideally it can automatically be added to an existing ontology and thereby enlarge the available ontological knowledge (for more details see section 2.3). However, for both information retrieval and extraction, there is a big piece of the puzzle missing:

> How can any natural language be related to semantic knowledge?
> How can computational processes use semantic information?

These are important questions as they are among the problems which prevent ontologies from reaching a broader coverage. This has also been pointed out by others: "Users who already have ontologies with rich instance data will benefit if they can make this data available to the IE components" [10]. It is easy to see that single words may relate to single ontological elements, e.g., the word "vehicle" relates to an ontological concept 'Car'. But what about compound words, e.g., "car tire" (which relates not to any tire, but to the 'Tire' of a 'Car') or proverbs (which mostly have metaphorical meaning)? How can morphological information (i.e., words whose appearance has been altered to transport additional information, e.g., the plural 's') be identified and matched to corresponding ontological elements? These are just some of the questions which remain largely unanswered in state-of-the-art approaches. The thesis therefore introduces the *Semantically Enhanced Domain Specific Natural Language (SE-DSNL)* approach, which offers solutions to those questions. The overall idea behind SE-DSNL was to develop a framework which allows domain experts to define how semantic knowledge and natural language can be combined. Texts which are available within this domain can then automatically be mapped to the semantic domain knowledge. The result of this process is a semantic interpretation of the text based on this knowledge.

The SE-DSNL concept is based on a generic and adaptable meta model. It captures both semantic as well as linguistic knowledge and provides detailed information about the relation between both of them. We show how computational processes can make elaborate use of the information at runtime by introducing concepts for pronominal anaphora resolution, word-sense disambiguation, vagueness and reference transfer. Additionally, a concept on how the extracted semantic knowledge can be used to retrieve specific texts is described.

The remainder of the chapter is structured as follows: Section 1.2 explains the problems which this thesis copes with in detail. Next, its objectives are specified in section 1.3. Afterwards, section 1.4 gives the reader an idea of what the SE-DSNL big picture looks like. Some parts of the thesis have already been published. All of these publications are presented in section 1.5. Finally, section 1.6 shows the outline of this thesis by giving a short description of every chapter.

1.2. Problem Definition

This section gives an overview of the problems which the SE-DSNL concept handles. Basically there are five different challenges in this thesis. The first one is how ontological and linguistic knowledge can be combined. The second deals with the standard pipeline architecture which is mainly used in natural language processing. Following, the third challenge describes how textual and ontological knowledge can be aligned. The fourth problem is that several different linguistic phenomena require semantic knowledge in order to be computed. The last challenge is about how a semantic interpretation can be classified based on its information. All these problems are described in detail in the following sections. For more insight into the different topics we refer the reader to section 2.

1.2.1. Ontologies and Natural Language

Ontologies are one way how the knowledge of a domain can be captured in a machine understandable way. A lot of research has been put into the formal specification of ontologies in the last years. The most important and de-facto standard for ontologies is OWL (the current version is OWL 2 [11]). Its focus lies on the semantic formalities and reasoning capabilities. However, one thing that became clearer during the past couple of years is that for a better and more widespread distribution of OWL a better way to handle OWL ontologies and the knowledge within them is necessary. For one, creating ontologies is difficult because of understanding logical formalisms. Further, state-of-the-art ontology design tools have unintuitive user interfaces which add additional complexity to the development of ontologies. One possibility to solve the problem could be an automated information retrieval and extraction process from natural language text. However, a clear bridging between the unambiguous ontological knowledge and highly ambiguous natural language is missing. The problem starts at the word level (a single word can have multiple senses) and continues up to the syntactic level (sentences can have more than one meaning). Further, different languages have different ways of representing meaning (see section 2.1). Some approaches are available on how this gap could be closed, e.g., LexInfo [12]. These, however, neglect certain linguistic phenomena and are therefore only applicable to specific languages.

Problem In order to use ontologies more efficiently they need to be associated with linguistic information. Pure semantic information are not enough to allow a broad usage of ontologies as the problems of knowledge acquisition and usability remain. Both problems often rely on natural language text which is difficult to map to semantic knowledge

especially if there is no information available on what this mapping looks like. Therefore, ontologies should contain information how natural language can be mapped to ontological knowledge in order to enhance the precision of automated information extraction and retrieval.

Challenge In order to better integrate ontological knowledge into Natural Language Processing (NLP) related tasks it is necessary to specify how ontological structures can be represented in a specific natural language. The approach must be flexible enough to represent the morphological and syntactic details of any natural language as well as the semantic knowledge of any possible domain. Still, this information should be created in a way which makes it possible to use it in computational processes.

1.2.2. Pipelined processing

More and more applications require the analysis of natural language, often referred to as Natural Language Processing. NLP covers everything that has to do with the computational analysis of either spoken or written natural language. Each of them can be divided into many more subproblems. The thesis focuses on written text and especially the interpretation of the semantic information that it 'transports'. Today most components which care about the analysis of natural text are concerned with the analysis of the syntactic part only. It poses different problems like

1. Sentence Splitting: The problem of dividing a single text into its single sentences. It may first seem as a simple task because the end of a sentence is most commonly marked by a period. However, a period can also be used to indicate an abbreviation ("i.e., ") or to mark the decimal point in a real number (at least in English culture). This makes it somewhat more difficult to correctly retrieve the single sentences. This poses therefore a first possibility to introduce ambiguity, i.e., different possibilities to interpret a sequence of characters.

2. Tokenization: The problem is similar to sentence splitting. However, instead of splitting a complete text into sentences, a single sentence is separated into its single tokens (words, punctuation marks, etc.). Still, this also represents an opportunity to introduce new ambiguities, e.g., "i.e., " could be divided into four single tokens "i", ".", "e" and ".", although it should be treated as one single token.

3. Part-Of-Speech (POS) Tagging: POS tags describe the grammatical function of a word within a text, i.e., if a word is a verb, a noun or something else. POS tagging therefore refers to the process of assigning the correct POS tag to a corresponding word. The problem is difficult as the POS tag of a word normally depends on the

context of its surrounding words.

4. Syntax Tree Parsing: Based on the POS tags this task tries to create a tree like structure which is based on the grammar of the language. As there are often multiple possibilities to assign POS tags to a sentence there are also multiple possible syntax trees (often a standard sentence can yield thousands of possible parses, most of which, however, have no rational meaning).

These are some of the standard tasks that NLP has to cope with. Other challenges involve things like Named Entity Recognition (NER), pronominal anaphora resolution and Word Sense Disambiguation (WSD), etc.

Problem The standard architecture for NLP today is a pipeline, i.e., all of these previously mentioned tasks are executed by clearly distinct components, one after another. The reason is that each NLP component requires the information of a previous component, e.g., the syntax tree parser requires each word to have its POS tags assigned. Hence, these must first be created by a POS tagger. The approach is rational from a computational point of view. Treating a single problem is easier and leads to less complex analysis components than treating several problems at once. Still, the amount of ambiguities can not be limited as additional information might be necessary. The information, however, might only be available in a later analysis step of the pipeline. Therefore, as one component returns an erroneous result which is used as the input for another component this new component can most likely not produce a good result either.

Challenge Many of the NLP related tasks have to handle difficult problems like ambiguity, which require different types of information. The core of the thesis is the interaction of semantic and linguistic knowledge. Especially semantic information are very valuable within natural language processing. In order to provide NLP specific components with an optimal pool of information they must have access to information of other NLP components and vice versa. A pipelined approach can not support the required degree of interaction between different components. Hence, another type of architecture is required.

1.2.3. Alignment of Ontology and Natural Language Text

Some companies face the problem of having multiple sources with the same information, however, there is no direct mapping available between them. This is especially the case for ontological information sources on one side and natural language documents on the other. The first type gives computers the power of automatic reasoning whereas

the second source is much easier for humans to understand and update. Normal search engines are primarily based on retrieving documents which match a specific keyword based search query, i.e., documents are matched against the keywords. If the keywords can be found within a document, this document is returned. Such a process works well to a certain degree. Similar techniques have been used to match ontologies to documents, i.e., by simply searching words within the documents which can be matched to ontological elements. However, matching documents and ontologies this way does not yield good results because contextual as well as structural information should also be considered during the process. This is, however, difficult because (as pointed out previously) current ontology standards have no possibility to specify how semantic information can be mapped to linguistic knowledge. Therefore, ontologies and documents must currently be mapped manually, which can be a very time consuming task. Also, if the documents or the ontologies will be updated, all links might have to be updated as well. This means that every existing link has to be checked for its validity. Further, new ones have to be inserted if new content is available within either the documents or ontologies.

Problem There are many different approaches available which use ontologies for the semantic annotation of documents or information retrieval (both are tasks which present mappings between a natural language document and an ontology), however, none of those are based on a well specified mapping between semantic and linguistic information. Today, mappings between natural language documents and ontologies are mostly created either algorithmically or by using more shallow linguistic knowledge bases like WordNet [13] [14]. This, however, leads to imprecise mappings, which further lack deeper semantics. The reason is that algorithms are not suited to represent the linguistic knowledge or how this knowledge can be mapped to an ontology. Additionally, linguistic knowledge sources like WordNet have no well-specified mapping available domain ontologies and also do not contain information about the grammar of a given language.

Challenge The challenge is to use well defined mapping information between ontological and linguistic knowledge to align natural language texts to a given ontology. The process should not only map natural language on a single word level, but also on its syntactic level. Furthermore, the knowledge within the ontology should be used to enhance the alignment result, e.g., by incorporating the semantic knowledge into the resolution of linguistic phenomena like pronominal anaphora resolution.

1.2.4. Ambiguity, Vagueness and Reference Transfer

Natural language contains many phenomena all of which are difficult to handle in computational linguistics. Many of those require deeper meaning in order to fully understand them. Some of those phenomena are ambiguity, vagueness and reference transfer. Ambiguities in general describe situations in which a specific type of information can have multiple interpretations. The most prominent type is probably word sense ambiguity, i.e., one word with multiple meanings. Vagueness specifies words whose meaning is not exactly specified (e.g., someone who talks about a "car" in general although actually meaning his very own car). The third phenomenon, reference transfer, means situations in which a word receives a completely new meaning. Humans tend to make use of those phenomena without even noticing. Most of the time the listeners / readers do not have problems in understanding what the author is referring to, because they have knowledge as well as experiences of their own. All three phenomena are important for computational linguistics. If they are not treated accordingly the correct concepts for specific words or terms may not be identified.

Problem The problem with most concepts for treating ambiguity is that they can only identify the best matching concept out of a set of existing concepts. Hence, as humans tend to overgeneralize concepts (i.e., refer to an entity by using a very generic term, e.g., referencing one's own car with the word 'Car' instead of using the term which correctly identifies the specific car type), more specific information may not be part of the initial concept set. Further, vagueness and, especially, reference transfer are rarely being treated in computational linguistics at all because the identification of the correct concept can not be done based on a simple mapping approach, but requires a certain degree of reasoning in order to identify the most probable concept. This means that based on a set of available information new information must be identified which matches a specific situation.

Challenge The SE-DSNL concept is based around a domain specific ontology which can be represented linguistically. The semantic information within the ontology should be made available such that it helps to solve the aforementioned linguistic phenomena by using some type of reasoning mechanism.

1.2.5. Retrieving Information from Semantic Interpretations

A semantic interpretation of natural language text provides a multitude of precise semantic information, especially in comparison to a pure syntactic representation. For example, the senses of the single words as well as the connections between the available concepts

can be used for several different applications, e.g., search engines or the automatic classification of natural language texts. Both are of importance in different scenarios, e.g., in end user support departments texts should automatically be forwarded to the responsible person depending on the content of the text. Another scenario can be applications which should be controlled by using natural language commands. Standard syntactic based approaches can have problems with this task because of ambiguities as well as unknown semantic relations. A semantic interpretation of text can greatly enhance the precision of the process, because it should contain both the semantic and structural information.

Problem Classical approaches rely, as previously mentioned, on syntactic features, i.e., word stems, perhaps syntactic trees, word frequencies, etc. Semantic approaches can enrich the words within texts with additional semantic information based on, e.g., word sense disambiguation. However, not only the meaning of single words is important, but also the semantic relations between the concepts of the words. Therefore, both classification as well as information retrieval processes, which rely on the semantic interpretation of text, need to base their mechanisms on the semantic types as well as the relations between those concepts. Such a process must be precise in order to use the available information accordingly. However, it also must exhibit a certain degree of tolerance. This is due to the fact that two semantic interpretations can describe the same content but still vary in their specificity of the mentioned semantic elements as well as the structural relations between them. Hence, the approach must be tolerant to both of these challenges.

Challenge The challenge is to develop a concept for the challenges of information retrieval and classification. The approach must allow the definition of tolerant and still precise constraints about both the semantic elements and structure between them in the semantic interpretation. Ideally, one set of constraints can be used to retrieve and classify information from different natural language descriptions.

1.3. Objectives

In the previous section, the problems which SE-DSNL has to cope with have been presented. In this section the objectives and approaches of the thesis are described in detail.

1.3.1. Combining Ontologies with Natural Language

In the previous section it was mentioned that current ontology standards lack the capability of associating semantic knowledge with detailed linguistic information. In order to use ontologies in NLP related scenarios this is a necessary prerequisite. Therefore, the thesis describes a concept of how ontological information (i.e., its concepts as well as the relations between the concepts) can be mapped to linguistic information.

Approach We develop a concept which overcomes the linguistic limitations of OWL. The concept provides a generic way of mapping every possible linguistic form and structure to a corresponding concept or structure within the ontology. This is done by allowing the integration of additional external functions which can be tailored to best fit language specific phenomena. Such functions have access to both the semantic as well as linguistic information of an SE-DSNL ontology.

Contribution A concept as well as a meta model which specifies how ontologies can be enriched with linguistic information.

1.3.2. Concurrent Analysis of Natural Language Specific Problems

As it was mentioned before state of the art NLP mostly relies on sequential approaches, i.e., components for different problems are lined up within a pipeline and use the output of one component as the input for the next. In this thesis one objective is to create a concept which allows the concurrent treatment of different NLP related problems. This means that one analysis component A should have access on the results of a completely different analysis component B and vice versa. In case that B creates new results, component A should be capable of reevaluating the new information.

Approach We develop a concurrent and generic approach which parses a given input (i.e., a written text) in multiple iterations until no more results can be deduced, i.e., a stable result set has been reached. The approach is easily extensible and allows components

to communicate with each other, i.e., one component delivers new information which are used by another component that can make new contributions etc. The parsing process is based on the information available within the meta model of objective 1.3.1.

Contribution A generic concept as well as prototypical implementation for the concurrent analysis of natural language specific problems.

1.3.3. Semantic Interpretation of Natural Language Text

Previously the problem of synchronizing natural language documents with ontologies has been mentioned. One objective of the thesis is therefore, to create a semantic interpretation of a given natural language text. The interpretation must contain the ontological concepts which the words of the text refer to. Also, the relations between the concepts should be identified as they are stated within the text. The interpretation must adhere to the mapping between ontological knowledge and linguistic information as described in section 1.3.1.

Approach We integrate the contributions of objectives 1.3.2 and 1.3.1. This leads to a consistent framework which concurrently analyzes text, using the ontology as both a source of information as well as an anchor for the semantic interpretation. Because of the external functions, which have been specified as part of the meta model, the knowledge of the ontology can be used at runtime to further enhance the overall parsing precision and result, e.g., by resolving pronominal anaphora.

Contribution A consistent framework, combining the contributions of objectives 1.3.2 and 1.3.1. The framework is further evaluated on the basis of two case studies.

1.3.4. Handling Ambiguities, Vagueness and Reference Transfer

As shown previously in section 1.2.4, coping with ambiguities, vagueness and reference transfer is important in order to correctly identify the meaning of specific words. It is therefore one objective of this thesis to cope with word sense disambiguation, vagueness and, to a certain degree, reference transfer, i.e., based on the ontology, the correct meaning of the words should be identified. Further, generic word to concept mappings should be made as specific as possible, if very vague words have been identified. Also, if an object is referred to by using just a property the correct concept should still be found.

Approach In similar scenarios in the past, spreading activation based approaches have been used if ontological knowledge was available. In order to solve the previously mentioned linguistic phenomena, a similar approach has been considered here, which incorporates a semantic distribution of the tokens as well as further heuristics to identify the correct information.

Contribution The contribution is an ontology-based algorithm which helps in natural language scenarios that include ambiguities, vagueness or reference transfer.

1.3.5. Semantic Information Retrieval

In section 1.2.5, the problem of retrieving information from semantic interpretations was introduced. Hence, one objective of the thesis is the development of a concept which allows us to retrieve specific information from a semantic interpretation of a natural language text. Further, the approach should be usable to classify an interpretation according to certain criteria, i.e., based on these criteria and the concepts of the words as well as the relations between those concepts.

Approach Experts can define so called patterns, i.e., a pattern represents a category and contains a definition of specific semantic information which must be contained within a semantic interpretation. Next, a graph matching algorithm matches the pattern against the given interpretation. If the pattern can be applied to the interpretation, the interpretation is classified as an element of this specific pattern category. Moreover, the elements of the pattern are assigned to corresponding elements of the interpretation, thereby allowing the retrieval of specific information.

Contribution The contribution is a concept and a prototypical implementation which retrieves information from semantic interpretations according to pattern-based specifications.

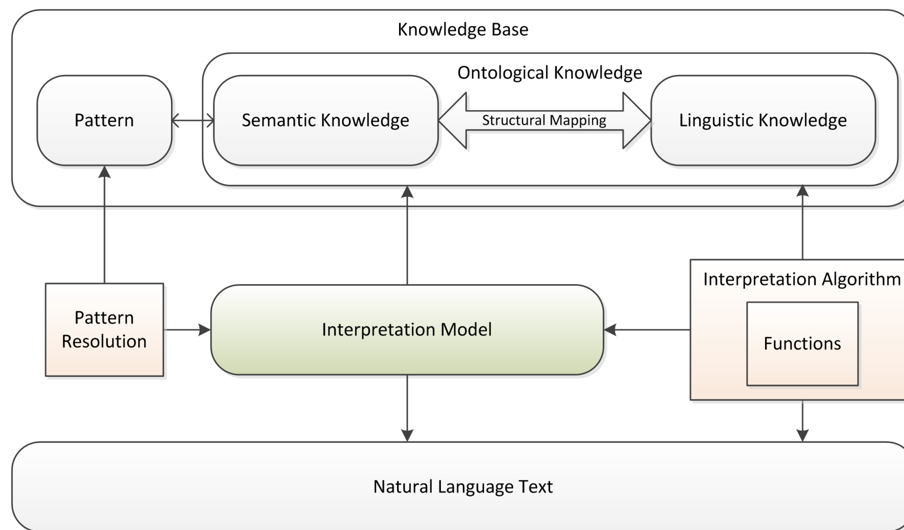


Figure 1.2.: Architecture of the SE-DSNL approach

1.4. Architecture

In order to solve the previously specified objectives we developed the architecture presented in figure 1.2. It is based on a knowledge base which captures all the relevant information and allows them to be interrelated. This means that semantic knowledge is connected to linguistic knowledge both on an element as well as a structural level. It allows the representation of phenomena like homonymy and synonymy, as well as capturing arbitrary grammatical relationships. The information is used by an interpretation algorithm for parsing a natural language text. The algorithm parses the text by applying user specified functions. These can be tailored to the exact needs of a given domain and work on both the linguistic as well as the semantic level at the same time. They thereby provide an optimal basis for analyzing natural language text. The result of the parsing process is stored within an interpretation model, which captures the exact relations between the natural language text on one side and the ontological knowledge on the other. Hence, it stores all information about which word or phrase corresponds to which semantic element. In order to retrieve information from this interpretation model, a pattern based approach has been developed which allows the definition of semantic patterns by directly associating them to the semantic knowledge of the knowledge base. The information is used by the pattern resolution component, which tries to match all available patterns on a given interpretation model.

1.5. Publications

Different parts of the thesis have been published in the past. The section gives an overview of these papers and shortly describe how they are related to this thesis.

- [15] Wolf Fischer and Bernhard Bauer. *Cognitive-Linguistics-based Request Answer System*. In: *AMR 2009 - 7th International Workshop on Adaptive Multimedia Retrieval, Madrid, Spain, September 2009*: The paper introduces a first version of the meta model, presented in section 3 and further gives an outlook on how the model might be interpreted.
- [16] Wolf Fischer and Bernhard Bauer. *Domain Dependent Semantic Requirement Engineering*. In: *DE@CAiSE'10 - Workshop on Domain Engineering, Hammamet, Tunisia, June 2010*: The paper gives an overview on how SE-DSNL might be used for the task of requirements engineering.
- [17] Wolf Fischer and Bernhard Bauer. *Combining Ontologies And Natural Language*. In: *AOW 2010 @ AI 2010, Adelaide, Australia, December 2010*: The paper presents the final version of the meta model in section 3.
- [18] Wolf Fischer and Bernhard Bauer. *Ontology based Spreading Activation for NLP related Scenarios*. In: *SEMAPRO 2011, Lisbon, Portugal, November 2011*: The publication describes the spreading activation based algorithm from section 5, which treats WSD, vagueness, reference transfer and further identifies the semantic relatedness of different semantic elements.

1.6. Outline

The thesis is structured as follows: Chapter 2 gives an overview and introduction into all the different topics which this thesis covers. It consists of three main sections: The first section 2.1 introduces the basics of natural language, its challenges and how all of these are related. The next section 2.2 describes ontologies, relevant standards and also one of the most advanced concepts to linguistically ground OWL ontologies. The final section 2.3 covers the basics of information retrieval in general and ontology-based information retrieval in specific.

The next chapter 3 introduces the SE-DSNL meta model which captures the information of how ontological structures can be represented with linguistic information. It therefore first presents a set of specific requirements in section 3.2. We developed a meta model which fulfills all of them. Its elements and structure is shown in section 3.3. A formal specification of the meta model is given in section 3.4. Further, section 3.5 describes how knowledge for this model can be imported from existing OWL ontologies. It also specifies the process of combining semantic with linguistic knowledge as well as specific modeling guidelines.

Chapter 4 covers the concept of how the information within a SE-DSNL model can be used to interpret natural language text. After a short introduction (section 4.1) the process of analyzing text and creating an interpretation from it is introduced in section 4.2. Next, section 4.3 describes a set of different functions for parsing textual information. The chapter is concluded in section 4.4 which represents related work and delimits it from SE-DSNL.

Chapter 5 introduces an algorithm which is used for retrieving information from the semantic part of a SE-DSNL model. The chapter first contains an introduction to the different tasks of this algorithm (section 5.1), before it presents its specific requirements (section 5.2). After a set of required definitions (section 5.4) and the description of an ongoing example (section 5.3), the three phases of the algorithm are explained in sections 5.5, 5.6 and 5.7. We show that the algorithm terminates and the results are valid in sections 5.8 and 5.9. The chapter is concluded by delimiting our approach from others in section 5.10.

The following chapter 6 explains how information can be retrieved from a semantic interpretation by using a pattern-based approach. Section 6.1 introduces and motivates the chapter. It further gives a detailed definition of the problem and the requirements the approach has to fulfill. Next, the required extension of the SE-DSNL meta model is presented in section 6.2. Also, the semantics of the model are explained. The mechanisms of the algorithm which classifies an interpretation model is shown in section 6.4. Section 6.5

concludes the chapter by delimiting the concept from other existing approaches.

Chapter 7 presents the results of the evaluation. It first defines the criteria and methods used to evaluate the concept of this thesis in section 7.1. Next, the prototype which realizes the different concepts is described in section 7.2. How well the SE-DSNL concept can handle challenges like modifiability, reusability and performance is shown in section 7.3 by using different scenarios. The first case study, described in section 7.4, shows how well linguistically complex domains can be handled with SE-DSNL. In contrast, the second case study (section 7.5) focuses on high parsing precision and the retrieval of specific information.

The final chapter 8 concludes the thesis. It gives an outlook on potential future work and applications in section 8.2, before section 8.1 summarizes the experiences and results of this thesis.

The thesis is best read following the approach which can be seen in figure 1.3. Starting with the introduction (and the basics, optionally), the reader should continue with the chapters about the combination of ontological and linguistic information, the semantic interpretation of natural language and the semantic spreading activation. After these chapters, one can decide, based on his / her interest in semantic information retrieval. If the topic does not interest the reader, one can directly continue with the first case study and, afterwards, go to the conclusion chapter. If one intends to read the chapter about semantic information retrieval and classification, the reader will have acquired the necessary knowledge which is required to fully understand the complete evaluation chapter.

One final note: From time to time some of the words within this thesis start with a capital letter, e.g., Function, Construction etc. These words refer to elements from our meta model and are capitalized to recognize their origin more easily.

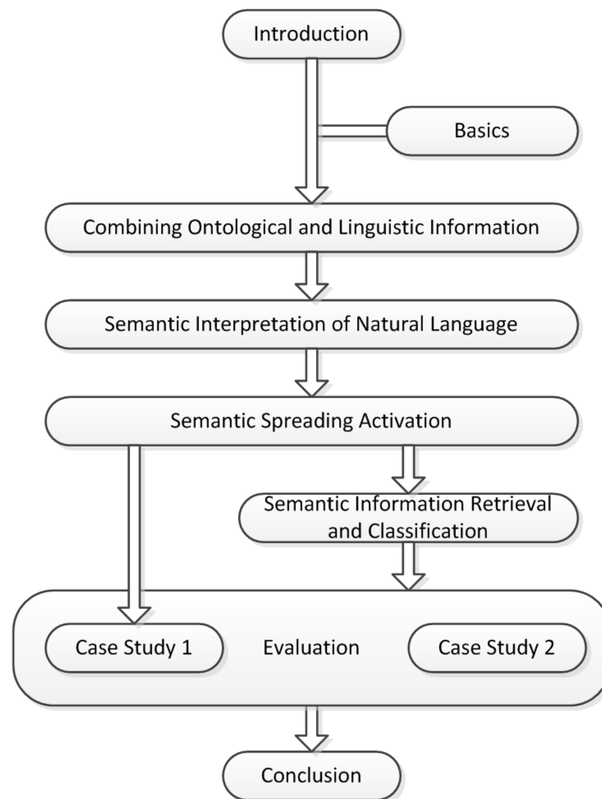


Figure 1.3.: Outline of the thesis

2

Basics

2.1. Natural Language

An important area of this thesis is natural language, which contains a variety of phenomena that make a mapping to semantic information challenging. The area of research that tries to understand natural language, its history as well as evolution and the way it is used for communication, is called linguistics [19]. Linguistics is an important field of research, especially because it also tries to uncover the commonalities between different languages, even between such seemingly different ones as, e.g., German and Chinese. This is called the *unified phenomena*, i.e., "linguists assume that it is possible to study human language in general and that the study of particular languages reveals features of language that are universal" [19]. The assumption is important: Its absence could mean that different languages can only be mapped to an ontology by using different approaches and concepts. In the following, we give an introduction to the field of natural language and present its structure, problems, challenges and approaches which are relevant to SE-DSNL. Those sections which cover linguistically related basics do not focus on one language, but instead describe the commonalities of all known languages.

2.1.1. Morphology

Morphology is concerned with the study of words, i.e., how words are constructed, what smaller subunits they are made of, etc. The smallest subunits, a word consists of, are so called morphemes [20] [21]. A morpheme is the smallest unit which can contain meaning. An example is the word "dog", which is only one morpheme. In contrast, "cats" consists of two morphemes, one being the morpheme "cat" and the other one being the plural "-s". Morphemes therefore can express very different types of meanings, either of a concrete (e.g., "cat") or a more abstract kind (the plural "-s", past or future tenses, etc.). Morphemes are mostly distinguished in two main classes: *stems* and *affixes*. Basically, a stem morpheme holds the main meaning of a word, whereas affixes "manipulate" the meaning of the stem. Affixes themselves can be divided in further subcategories [20]:

1. Prefixes, i.e., coming before the stem. Example: The German word "unschön" (English: not nice), which is composed of the prefix "un" and the stem "schön".

2. Suffixes, i.e., coming after the stem. Example: The German word "sagt" (English: Says), which is composed of the stem "sag" and the suffix "t".

3. Circumfixes, i.e., both coming before and after the stem. Example: The German past participle word "gesagt" (English: said), which is composed of the prefix "ge", the stem "sag" and the suffix "t".

4. Infixes, i.e., inserted into the stem. Example: In the Philipine language Tagalo, the

word "humingi" (English: borrow) is composed of the stem "hingi" and the infix "um" (which marks the agent of the action).

There are four ways of how a word can be built:

1. Inflection: The stem of a word is combined with a grammatical morpheme. This normally does not change the class (POS in section 2.1.6.1) of the word, but its grammatical function (e.g., indicating the tense when something happened).
2. Derivation: Similar to inflection, the stem of a word is combined with a grammatical morpheme. This time, however, its class changes, often also leading to a different meaning. For example, the verb "computerize" can take the affix "ation", which leads to "computerization".
3. Compounding: Here, multiple stems are combined to form a new word. For example, the word "Mousecable" is the combination of the two words "Mouse" and "Cable".
4. Cliticization: A stem is combined with a *clitic*, i.e., a morpheme which behaves as a word but is reduced in size and attached to another word. An example is the English expression "I've", in which "'ve" is the clitic of the word "have".

As can be seen, the meaning of a word depends on the stem and the affixes that are used to build it. Depending on the language, different ways of creating words are preferred over others [21] [22]:

1. Analytic languages: Words consist of only one morpheme, i.e., the stem of the word. They do not make use of any type of inflection. Examples are Chinese, Vietnamese.
2. Synthetic languages: In contrast to analytic languages, synthetic languages make use of inflection, derivation and compounding. Words in such languages therefore consist of more than one morpheme. Languages, which make extensive use of morphemes, are called polysynthetic (e.g., American Indian languages, Eskimo).
3. Agglutinative languages: This is a subtype of synthetic languages. An agglutinative language ideally expresses three properties: Each morpheme represents only one meaning, morphemes are clearly separated and grammatical properties do not affect the form of the individual morphemes. A good example is Turkish.
4. Flective languages: The category is also a subtype of the synthetic languages and represents the opposite of the agglutinative languages, i.e., one morpheme represents more than one meaning, several morphemes can be merged into a single morpheme and grammatical properties can affect the form of individual morphemes. An exemplary language is Indo-European.

It has to be noted that languages do not necessarily fit into one of these classifications directly. Instead they may have soft borders and, therefore, overlap with several of the categories. As can be seen, there are many different ways of creating words in different languages such that they transport the meaning which was intended by the author. The variety, however, is a problem if trying to represent semantic knowledge. This is why a concept must be capable of handling all different kinds of expressing meaning. We developed SE-DSNL in a way that it can be adapted to represent semantic knowledge in any language.

2.1.2. Syntax

Syntax (coming from the Greek word *sýntaxis* which means "setting out together or arrangement" [20]) "is the study of the principles and processes by which sentences are constructed in particular languages" [23]. First work on syntax can be dated back to Panini who wrote a book about the grammar of Sanskrit [24], which is still used in teaching Sanskrit today [20]. From the 17th century on, most researchers believed that there is a universal grammar that can express any possible thought, as thought processes were believed to be the same for every human [25]. The idea was introduced by Arnault and Lancelot in 1660 [26] [27]. However, the approach was based on the French language only, therefore contradicting the "universal" idea. It took until the 19th century, in which the raise of comparative methods started to diminish the "universal" thought [25]. The most important recent advances in linguistics were made in the 20th century, most notably because of Chomsky [23]: "Chomsky is currently among the ten most-cited writers in all of the humanities [and social sciences] (behind only Marx, Lenin, Shakespeare, the Bible, Aristotle, Plato, and Freud) and the only living member of the top ten" [25]. He thought that as humans are capable of producing an infinite variety of sentences from a more or less finite set of words, there must be some kind of innate competence to generate those sentences. This lead to the term "Generative Grammar" [25]. Chomsky believed that this competence can be expressed in a formal way by using rules which allow the construction and understanding of a nearly unlimited number of sentences. We refer to such a set of rules also as grammar [28]. In the following, the most basic building blocks of formal grammars and how they can be used to construct more complex sentences are explained.

2.1.2.1. Syntactic Categories and Constituents

Before introducing the different building blocks of language we first introduce another important concept of language, i.e., the so called Part-of-Speech classes (also called *syn-*

tactic categories). Each word within a sentence has its own POS class which specifies its grammatical properties [28]. For example, the POS class *VERB* represents such words which represent an action or event. A *NOUN* is a word which specifies names, people, locations, etc..

A man draws a picture.

In the example, both "man" and "picture" are of the category *NOUN*, whereas "draws" is a word from the category *VERB*. An important aspect is that the set of syntactic categories is not the same for every language but instead can differ greatly. Nearly every language contains verbs and nouns [29] and, therefore, also a corresponding POS class. But beyond this, there can be huge differences between languages. For example, Japanese has four different kinds of adjectives [30], whereas in the English language there is just one. As it was already mentioned, grammars use rules to combine the single words into a sentence. Basically, several words within a sentence can be grouped together to form a subgroup, also called *constituent*. Constituents themselves are part of a hierarchical order. They can be differentiated into *phrases* and *clauses*, where phrases are lower in the hierarchy than clauses [28]. Most phrases have a so called *HEAD*, i.e., a word which contains the core meaning of the specific phrase. As it is the case with single words, phrases also have a syntactic category, therefore representing a specific grammatical function. Most often the *HEAD* word will "be a lexical item of the same category" [28], i.e., the head of a noun phrase is a noun, the head of a verb phrase is a verb etc. An example can be seen in the following:

A man draws a red picture.

In the example, the words "A man" as well as "a red picture" form a phrase, more precisely a *NOUNPHRASE*. In the second *NOUNPHRASE*, "picture" is the *HEAD* of the phrase with the word "red" being a so called *DEPENDENT* of the word "picture" (as it modifies / specifies the meaning of the word "picture"). The corresponding hierarchical order can be seen in figure 2.1. Note that the node 'S' is the clause of the sentence, indicating the grammatical function *SENTENCE*.

2.1.2.2. Semantic Roles and Grammatical Relations

Previously, we introduced the concepts of syntactic categories, constituents and its specializations phrases and clauses. We further showed how these different concepts belong together. So far we only know what sentences are made of and that each of these part has a grammatical function which is represented by a syntactic category. The most important idea of syntax, however, is how these different parts and concepts together give a meaning to a sentence - that is the question behind semantic roles and grammatical relations.

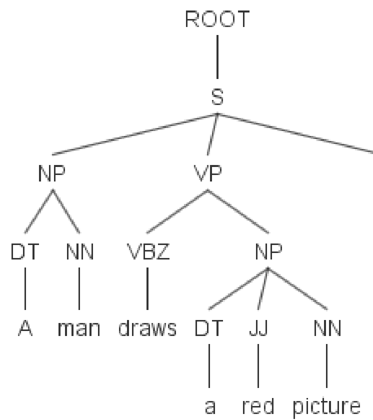


Figure 2.1.: An example of a constituent tree, which has been generated with the Stanford Parser 2.0.1 and the English PCFG caseless model

Let us have a look at an example:

> He met his friend.

Each grammatically correct sentence has a so called *predicate* which contains the meaning of the relationship of a sentence. In the previous example, that is the word "met" (in more complex sentences, the predicate can also span multiple words). A predicate is either true or false with regard to its so called *arguments*, i.e., "the individuals (or participants) of whom the property or relationship is claimed to be true" [28]. Here, the word "met" has two arguments "He" and "his friend". The predicate, together with its arguments, is called a clause (see section 2.1.2.1). Depending on the verb which is used within a clause and the context of the sentence, the number of arguments can vary. For example the verb "run" can be used with a number of different arguments:

> She opened the door.
> The door was opened.

The first sentence simply indicates that a women opened a door. In the second sentence, a door is being opened. Both are valid sentences (both syntactically as well as semantically), in which the verb "open" receives a different number of arguments (in the first sentence, there are two arguments and in the second sentence, there is only one). The number of arguments of a verb is also being referred to as *valency* [20] [21].

We know now that there are predicates and each predicate has a certain number of arguments. Interesting to see is that in a standard sentence there is almost always someone doing something to somebody. Linguists therefore developed the concept of *semantic roles* (sometimes also called thematic role) [31] [32]. A semantic role basically is a broader classification of the different arguments used by a predicate. In the previous example, "She"

would be a so called *agent* (i.e., someone who is doing something) and door would be a so called *theme* (i.e., "an entity which undergoes a change of location or possession" [28]). There are multiple additional semantic roles, e.g., *experiencer* (a living entity which experiences something), *instrument* (a non-living object which is used for doing something), etc. [28]. As it is the case with syntactic categories, there are multiple semantic roles, some of which are specific to certain languages.

Semantic roles help to identify the *predicate-argument structures* of a sentence, i.e., the relation between a predicate and its arguments. The reason is that no matter if the sentence is in an active or passive scheme (as long as it contains the same arguments), the predicate-argument structure does not change [28]. This can be seen in the following two examples:

> He programmed an application.
> An application was programmed by him.

Both sentences describe the same situation, i.e., a male person programs an application. The difference between both sentences is that in the first sentence an active verb form is being used, whereas in the second sentence a passive verb form describes the scenario. Still, the meaning remains the same. When children learn grammar in school, they often start with the so called *Subject (SUBJ)* and *Object (OBJ)*, two commonly used terms in linguistics. Knowing the SUBJ and OBJ of a sentence is important as these two define the main constituents of a clause and therefore help deciphering the complete syntactic structure and meaning of a text. A SUBJ is often described to be the agent of something, whereas the object denotes the receiver of an action. However, this definition is not accurate, which can be seen in the second of the previous two example sentences: There, the SUBJ is "application" which, however, is not the agent of the sentence. The definition of what a SUBJ and OBJ is depends on the language. In English, properties like the word order (e.g., the SUBJ normally comes before the verb and the OBJ after the verb), *agreement* with the verb (i.e., both share a set of common syntactic features, e.g., if the verb presents a plural form, the SUBJ most likely has to end with a plural "s") and others [28] [33] help to identify what the SUBJ and OBJ of a sentence are. The terms SUBJ and OBJ represent so called *grammatical relations*, which are defined based on "their syntactic and morphological properties" [28]. Note that there can be an additional object within a sentence, the so called *Secondary Object (*OBJ_2*)*. Again, depending on the language, there are different ways of determining which is the primary and which the secondary object. This can be solved using agreement (if there is agreement between the verb and one of the objects, the verb most likely agrees with the primary object), word order (the primary object often occurs closer to the verb) and others [28].

He gave [his friend]$_{OBJ}$ [a book]$_{OBJ_2}$.

In this sentence, both objects have been marked accordingly. Here, "his friend" represents the primary object, whereas "a book" represents the secondary object.

Besides the already mentioned ones, there exist other types of grammatical relations. One of them is the so called *Oblique Argument (OBL)*. In contrast to SUBJ and OBJ, an OBL is marked by a preposition in English. Basically, oblique arguments are said to be less important to the clause than a SUBJ and OBJ. In the following example, "for a small company" is an oblique argument.

He developed an application for a small company.

The different types of grammatical relations so far represent arguments which are closely related to the predicate. There, however, exists another element, which is not necessary in order to understand a sentence. However, it still transports information which is necessary for a listener or reader to understand a described situation. Such elements are called *adjuncts*:

He formatted his hard drive last night.

Here, the words "last night" form an adjunct. In contrast to oblique arguments, an adjunct can be deleted without losing any important information [28].

2.1.3. Meaning

Meaning is more than just the sense behind single words. Meaning not only exists at a single word level, but can also be constructed from single morphemes, words and constituents up to the sentence and even text level. All of this depends on the grammatical structures chosen by the author or speaker of a text. Meaning can come in many different shapes and forms, some of which can be seen in figure 2.2. We use many of these different varieties of meaning in our every day communication with other people. Some of these forms are so called *literally* meanings, i.e., things that are meant exactly in the way that they were said [19]. For example, someone might tell another person to leave the room by saying "Please leave the room". Here, the words exactly specify the intention of the author. However, the person of the statement could also have said: "The door is behind you" [19]. In this case, the speaker used a *nonliteral* way of expressing himself. If taken literally, the last example would contain an information about the location of a door. However, given a context of two persons standing in a room, one of them with his back to the door, the meaning of this statement is too obvious. Instead, the speaker implied non-literally that the other person should leave the room. Both, literal and non-literal, are varieties of meaning which are used by speakers to express information (i.e.,

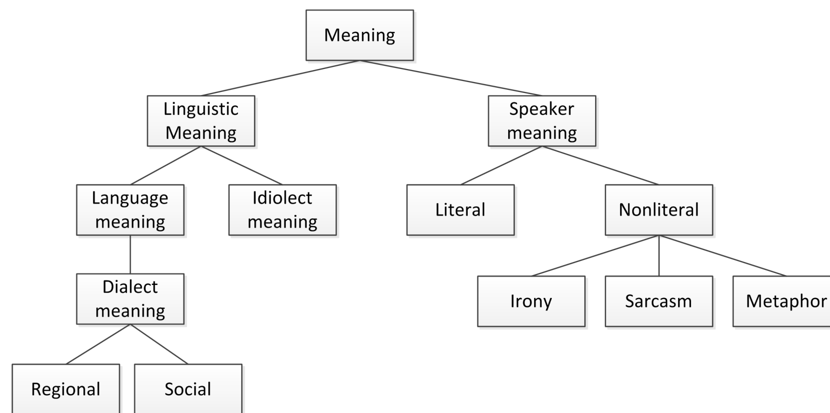


Figure 2.2.: Some varieties of meaning [19]

speaker meaning). Another form of meaning is the *linguistic meaning*. The form is concerned with the sense behind words or syntactic structures, e.g., the word "dog" refers to the (mostly) furry animal with four legs that sometimes barks. This form is very similar to literal meaning.

Now that we established that there are different varieties of meaning the question is what meaning actually is? Over the centuries, many different definitions have been proposed as to how meaning could be defined. However, none of these theories fully grasp every aspect that is known about meaning [19].

The denotational theory of meaning describes it as being the actual object that words refer to [34]. If, e.g., someone reads the words "red rose", the person most likely thinks of an actual red rose. There are, however, some shortcomings of this theory. For example would the approach mean that "if an expression has a meaning, then it follows that it must have a denotation" [19]. There are, however, many counter examples such as the simple words "the", "and", "hello", "Pegasus" (the flying horse) etc. which do not denote an object. The denotational approach is often taken by, e.g., truth conditional approaches [34], which try to identify the conditions under which the expressions of a given statement are true [20] [35].

Other theories which try to handle those problems at least partially, are conceptualist theories. In those the meaning of a word is not an actual object but an idea or concept within the mind of a person, a mental representation (e.g., Frege [36]). Although such theories can handle more abstract concepts like the one of "Pegasus", other abstract concepts are still difficult to grasp. Ontologies are one approach which makes use of this concept.

Pragmatic theories present another type of approach. These define meaning as the use that an expression has to the participants of an interaction [19] (therefore also called "meaning-is-use" theories [37]). The theory allows to specify the meaning of words like

"Hello". Its challenges, however, are that "the relevant conception of use must be made precise" and how exactly "the meaning is connected to use" [19]. Relevant to the thesis are mainly the conceptualist theories. The reason is that ontologies allow the definition of concepts, classes and properties in a machine-understandable way. They are therefore a kind of 'digital mind' which can contain a representation of knowledge.

2.1.4. Ambiguity, Vagueness and Reference Transfer

Language has a tendency to be *ambiguous*, i.e., a word or a syntactic structure represents several ways in which it could be understood [28] . Humans rarely have a problem with ambiguities, in contrast they often use ambiguities as an element of style for communication [38]. For a computer this is, however, a more difficult task as the process of disambiguation requires additional knowledge, as we show later.

The simplest form of ambiguity is the so called *lexical ambiguity*. Here, a single word has more than one meaning (such a word is called a *homonym*). An example is the word "bank", which can either refer to a financial institute or a river bank. Therefore, a sentence like the following one can theoretically have two different meanings [28]:

> A fishermen went to a bank.

So either the fisherman went to a financial institute or to a river bank. Another type of ambiguity is the so called *structural ambiguity*. Here, the problem is that different grammatical structures can be assigned to a set of words. This is best explained in the following example [28]:

> the [tall bishop]'s hat
> the tall [bishop's hat]

In the first phrase, the adjective "tall" refers to the bishop being a tall man, whereas in the second phrase the word "tall" describes the height of the hat. One area where humans use ambiguity as a linguistic style can be seen in humor. An example is the classic joke by Groucho Marx [38]:

> Last night I shot an elephant in my pajamas. How he got in my pajamas, I'll never know.

Here, the oblique argument "in my pajamas", referring to the predicate "shot", is ambiguous. The first interpretation refers to the author of the statement wearing pajamas, while he shot the elephant. Another valid interpretation for the "pajamas"-phrase would be to give further information about the location of the "elephant", which of course is highly unlikely. Still, the second sentence disambiguates the first one by specifying that Marx actually referred to the location of the elephant. Another type of ambiguity is pragmatic

ambiguity [38]. This type means that a complete statement is ambiguous because it lacks detailed information. The missing information has to be inferred by the recipient of the statement, e.g., by using information within his surroundings. Given a situation where one person wants to buy a new car at a car retailer, he / she might point to a specific car and say to the shop owner:

> I want this car.

Without the information to which car the buyer is pointing the statement is completely ambiguous, as it is unclear which car he is referring to. The shop owner therefore has to infer the remaining information from the situational context.

A phenomenon of language, which sometimes is mistaken for ambiguity, is *vagueness* [39] (the term is used especially in cognitive linguistics, whereas 'traditional' linguists refer to *generality*). A definition of the problem is: "A single lexeme with a unified meaning that is unspecified with respect to certain features" [38]. If for example someone talks to a car manufacturer and just mentions his "car", he *underspecified* what type of car he actually is referring to, although this might be of interest to the manufacturer.

One final phenomenon that we want to present here is *reference transfer* (sometimes also called *deferred reference*, *meaning transfer* or *sense transfer* [40] [41]) [38]. Here, the initiator of a statement assigns a new meaning to a word with an initially different sense. This can be seen in the following example:

> The ham sandwich is at Table 7. [41]

This is a quote from a server in a restaurant to one of his co-workers. He does not actually refer to the "sandwich" but to the person which ordered the sandwich at "Table 7". It is a very difficult problem as the mechanisms that lead to the phenomenon are "mysterious" [38].

All three phenomena which have been described so far, are important to the analysis of natural language. They are therefore also of importance for computational linguistics. In this thesis an algorithm was developed (section 5) which at least partially treats all of them.

2.1.5. Cognitive Linguistics and Construction Grammar

Cognitive linguistics present an approach which differs in many aspects from the traditional linguistic approaches, especially the one which Chomsky represented. Chomsky and his followers believed that language follows specific rules for which a specific part of the brain is responsible. This part of the brain was thought to be innate. In contrast, cognitive linguistics separates itself from Chomsky in three major points [42]:

1. There is no single part within the brain which is responsible for language alone. Language is thought to apply to basically the same concepts and mechanisms as any other cognitive process. Further, all linguistically relevant information (i.e., about meaning, morphology, syntax and phonology) is thought to be stored within conceptual structures.

2. Grammar is not about truth-conditional semantics (see 2.1.3), but about conceptualization. Humans experience situations and store them within their mind by conceptualizing them. The communication of an experience then depends on the grammatical properties that are chosen by the speaker. Persons receive and also communicate by using grammatical inflections and constructions which lead to different ways of expressing and perceiving the previously conceptualized knowledge.

3. Knowledge about language emerges from the use of language itself. The idea is that "categories and structures in semantics, syntax, morphology and phonology are built up from our cognition of specific utterances on specific occasions of use" [42].

Cognitive linguistics therefore does not analyze the different linguistic levels separately, but tries to look at them as a whole. Basically, language, communication and cognition "are mutually inextricable" [43]. This also means that meaning is inseparable from language which is the opposite of what Chomsky originally proposed [44]. In order to represent the inseparability for scientific analysis, the concept of Construction Grammar (CxG) has been developed. The approach allows combining forms with meaning directly. Many different approaches to CxG have been developed, all of which focus on different aspects. For example, Luc Steels developed the so called Fluid Construction Grammar (FCG) [45] [46] [47], which is used for researching the language learning and evolution. Another approach is the so called Construction Grammar by Ronald Langacker [48], which is concerned with the basic units of meaning and how they can be put together. Embodied construction grammar by Bergen and Chang [49] is used for the simulation of language understanding. Although all those approaches are different in how they are being employed, they also share some commonalities. First of all, the smallest entities in all of these approaches are *Constructions*. A construction is "the basic unit of linguistic knowledge to consist of form-meaning pairings" [49]. This means that there is no separation between syntax or semantics (also called the *syntax-lexicon continuum* [50] [51]). Also, all these approaches are based upon so called *unification*, i.e., the way how constructions are put together to form the meaning of sentences (see section 2.1.6.2 for more details). In the following, we present an example based on FCG [47] to illustrate the mechanisms and ideas.

As mentioned previously, cognitive linguistics has a holistic view at language. This is represented in FCG by having different rules for morphology, lexicography, semantic,

```
def-lex-stem "slide"
    ?unit
        referent: ?ev
        meaning: slide(?ev), slide-1(?ev, ?obj1)
                       slide-2(?ev, obj2), slide-3(?ev, obj3)
<–>
    ?unit
        form: stem(?unit, "slide")
```

Figure 2.3.: Example of a lexical stem rule [47]

```
def-sem slide-transfer-to-target
    ?unit
        meaning: slide(?ev), slide-1(?ev, ?obj1)
                       slide-2(?ev, obj2), slide-3(?ev, obj3)
<–>
    ?unit
        ?sem-cat: transfer-to-target(?ev), agent(?event, ?obj1),
                       patient(?ev, ?obj2), target(?ev, ?obj3)
```

Figure 2.4.: Example of a semantic categorization rule [47]

syntax and construction. An example of a rule which assigns a lexical word stem to a semantic structure, can be seen in figure 2.3. The rule defines that the stem "slide" is mapped to the referent "?ev" (variables are denoted by the prefix "?"). For this stem there is a predicate "slide(?ev)", which defines the semantics behind the referent. The other predicates with the attached numbers define additional arguments for the original predicate.

In a next step, the information from the lexical word stem rule have to be mapped into a so called semantic categorization which can be seen in figure 2.4. The rule maps the different, in figure 2.3 defined, arguments to their corresponding semantic roles (see section 2.1.2.2), i.e., "obj1" is the agent, "obj2" is the patient and "obj3" is the target.

All these rules are combined during the parsing process. For example, the semantic result of applying the rules of figures 2.3 and 2.4 as well as some others to a sentence like "Jill slides blocks to Jack" leads to the result in figure 2.5 (note that this is only a brief introduction and covers only the main parts of the semantic process; More information like the syntactic information can be seen in the corresponding paper by Steels [47]). There, the word "Jill" is mapped to the variable "?obj-20" in "unit2", "slides" to "?ev-9" in "unit3", "blocks" to "?obj-14" in "unit4" and "Jack" to "?obj-17" in "unit6". The structure is the result of a unification process in which the information of the different rules are combined to-

```
unit1
    sem-subunits: {unit2, unit3, unit4, unit6}
unit2
    referent: {?obj-20}
    meaning: {jill(?obj-20), status(?obj-20, single-object),
               discourse-role(?obj-20, external)}
unit3
    referent: {?ev-9}
    meaning: {slide(?ev-9), slide-1(?ev-9, ?obj-20),
               slide-2(?ev-9, ?obj-14), slide-3(?ev-9, ?obj-17)}
    sem-cat: {?transfer-to-target(?ev-9), agent(?ev-9, ?obj-20),
               patient(?ev-9, ?obj-14), target(?ev-9, ?obj-17)}
unit4
    meaning: {block(?obj-14), status(?obj-14, object-set),
               discourse-role(?obj-14, external)}
unit6
    referent: {?obj-17}
    meaning: {jack(?obj-17), status(?obj-17, single-object),
               discourse-role(?obj-17, external)}
```

Figure 2.5.: Semantic result for the sentence "Jill slides blocks to Jack" [47]

gether. This is, however, not yet the final result because the information about who is the agent, patient, etc. is not yet available. The result is created by applying a Construction which stores the information about how semantic and syntactic information have to be combined.

Such a construction can be seen in figure 2.6. The upper part before the two-headed arrow contains the semantic information, the lower part after the arrow the syntactic information. The construction here defines, how a subject-verb-object-to-object syntactic structure can be mapped on a transfer-to-target semantic structure. FCG is designed in a way that allows both the extraction of semantic information from text or the production of text from semantic information. If text is to be parsed, rules and constructions are being applied by checking the syntactic constraints and, if they match, "execute" the semantic part of the rules and constructions (the production of text works in exactly the other way). Putting all the rules and constructions together itself is based on a unification based approach, i.e., the features from two different rules are unified [46] [52].

We previously showed in a shortened example how construction grammars are used in a specific context. What all construction grammar based approaches have in common is the definition of constructions which themselves directly combine semantic and syntactic information. They further can reference each other to form complex meaning structures. The inseparability of syntax and semantics is important to the thesis and is one of the

```
def-cons transfer-to-target-construction
    ?top-unit
        sem-subsunits:
            ?event-unit, ?agent-unit, ?target-unit, ?patient-unit
    ?event-unit
        referent: ?event
        sem-cat: transfer-to-target(?event), agent(?event, ?agent),
                 patient(?event, ?patient), target(?event, ?recipient)
    ?agent-unit
        referent: ?agent
    ?patient-unit
        referent: ?patient
    ?target-unit
        referent: ?recipient
<–>
    ?top-unit
        syn-cat: SVOtoO-sentence
        syn-subunits:
            ?event-unit, ?agent-unit, ?patient-unit, ?target-unit
        ?event-unit
            syn-cat: predicate(?top-unit, ?event-unit)
        ?agent-unit
            syn-cat: subject(?top-unit, ?agent-unit)
        ?patient-unit
            syn-cat: direct-object(?top-unit, ?patient-unit)
        ?target-unit
            syn-cat: prep-object(?top-unit, ?target-unit)
```

Figure 2.6.: Construction for mapping a sentence, containing a sentence-verb-object-to-object syntactic structure, to a transfer-to-target semantic structure [47]

foundations for the concepts which is presented in the following chapters.

2.1.6. Computational Linguistics

This section introduces another important area of research, i.e., the area of computational linguistics. The field is concerned with the processing of natural language by computers. Its origins can be dated back to the midst of the past century. Some propose [21] that the start of it all was in 1949, when Warren Weaver suggested [53] that machines could be used to translate language. Since then, different directions in NLP have emerged [54] [21]:

1. Based on the work of Chomsky and his generative grammars [23], researchers started to elaborate how this can be applied to computational linguistics. This lead

to Tree Adjoining Grammar (TAG) [55] and HeadDriven Phrase Structure Grammars [56].

2. In contrast to generative grammars, statistical and datadriven concepts are based on extracting certain probabilities from large text corpora (i.e., machine readable text). The syntax of raw text can be analyzed based on these probabilities, lexical ambiguities can be resolved, etc. [21]. A prominent approach of this type can be seen in Artificial Neural Networks [57] [58] [54]. Due to their conceptual similarity to the physical structure of the brain the approach is considered to be a good match for Artificial Intelligence (AI) applications. Other advantages are tolerance towards noise and function approximation.

In the following different challenges of NLP are introduced which are relevant to SEDSNL.

2.1.6.1. Part-of-Speech Tagging

POS tagging describes the task of assigning the corresponding POS tags (as described in section 2.1.2.1) to the single words within a sentence. The process requires disambiguation, because words within a sentence can sometimes belong to different categories. For example, the word "book" can either be a noun (e.g., in the context of literacy) or a verb (e.g., describing the process of booking a ticket). Hence, a POS tagger has to incorporate the context of a word into its decision process [20]. Approaches to solving the task are either based upon rules application or statistical approaches, e.g., Hidden Markov models [59] [60]. One of the first taggers has been created by Eric Brill [61] and uses rules for solving the problem. One of the first Hidden Markov based taggers was CLAWS [62] [63]. Modern approaches use extensions of the Hidden Markov model like the Stanford NLP group [64] [65]. Modern parsers are capable of achieving very high precisions of up to 97.4% (as shown by [65], [66], [67] and [68]), however, these numbers can not necessarily be repeated in real life scenarios ([69], [70]). The reason is that the parsing input must contain the same characteristics as the original training data. Further, the precision value calculations are based on counting every single token and punctuation marks. A more accurate number could be the rate of correctly tagged complete sentences, which is about 55% to 57% ([70]).

2.1.6.2. Syntactic Parsers

Based on the POS tags it is possible to infer the syntactic structure of the sentence, which is referred to as either parsing or syntactic parsing. Different approaches have been presented in the past to this challenge. Earlier ones took a given grammar and tried to create

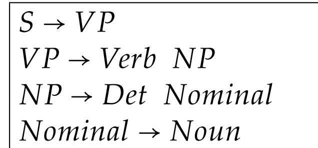


Figure 2.7.: Simple english grammar [20]

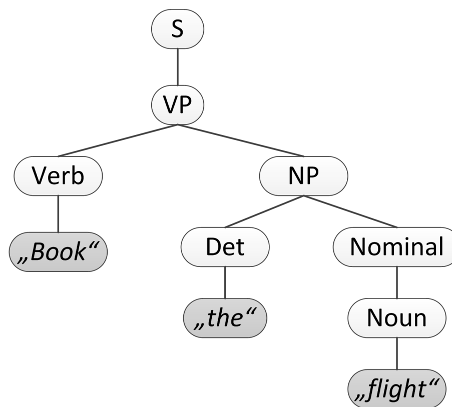


Figure 2.8.: Syntax tree of the sentence "Book the flight" [20]

a parsing tree in either a bottom-up or top-down manner [20]. A simple example for an English grammar and a syntax tree can be seen in figures 2.7 and 2.8. Applying the grammar from the first figure to the sentence in the latter figure results in the shown tree.

Modern parsers rely on statistical approaches using Probabilistic context-free grammar (PCFG). Here, each rule has a certain probability. The probability indicates how high the likeliness of a given non-terminal A is to be expended to a sequence β.

$$P(A \rightarrow \beta)$$

For all rules beginning with the non-terminal A the sum of the probability of all those rules equals 1.

$$\sum_{\beta} P(A \rightarrow \beta) = 1$$

The parsing process is therefore defined as finding the most likely syntax tree for a given sentence, for which different methods like *probabilistic CKY* [71] parsing exist. In order to assign probabilities to rules there are basically two different approaches [20]. The first one is based on a *treebank* (i.e., a collection of parse trees, e.g., the Penn Treebank [72]) and counts how often a specific expansion occurred within such a treebank [20]. If there is no treebank available, one can instead use an existing probabilistic parser which an-

alyzes the sentences and creates trees for them. Because of ambiguities in these trees and the problem that one needs a probabilistic parser in order to create probabilities for a probabilistic parser (a classic chicken-and-egg problem), the inside-outside algorithm was developed by Baker [73]. The algorithm works iteratively until the probabilities converge [20]. Another type of parsing has already been mentioned shortly in section 2.1.5, i.e., *unification* based parsing. For this type of parsing, *constraint-based formalisms* are required. The advantage of the approach is that it allows the representation of more complex linguistic phenomena as well as a detailed representation of information about number, gender, subcategorization, etc. [20]. In order to represent such constraint-based formalisms, one of the simplest and most commonly used approaches are *feature structures*, which basically consist of an *attribute-value matrix* (also often used in CxG). The examples in section 2.1.5 can also be seen as feature matrices, which are unified in a bottom approach. As the idea behind SE-DSNL is based upon construction grammars which often represent their information using feature matrices and unification, SE-DSNL also contains several of these characteristics in its parsing process.

There are different types of parsers available, i.e., constituency and dependency parsers. Constituency parsers (also called syntax or phrase structure parsers) create trees which represent the single constituents and phrases of a sentence. One of its most prominent candidates is the Stanford constituency parser. It can reach a precision of up to 89% ([74], [75]) if it has been trained correctly. In case that the default parser is being used on a completely unknown dataset the results can, however, drop significantly to about 50% [76]. In contrast, dependency parsers try to identify the dependency relations within a sentence by using a dependency grammar. Their precision ranges from similar values like constituency parsers to lower scores as shown by Cer [75].

2.1.6.3. Word Sense Disambiguation

Word Sense Disambiguation describes the process of identifying the correct sense for a word with multiple senses (as described in section 2.1.4) [20]. First approaches to this problem were made by Katz and Fodor in 1963 [77] as well as Wilks in 1972 [78], both of which made use of "hierarchically organized selectional constraints with complex semantic representations called formulas" [21]. The assumption was that the meaning of each word was expressed by one such formula. For example, the formula for a verb contained semantic information about the arguments it was associated to. The disambiguation was carried out by selecting the formulas which lead to as much "overlap" as possible between the formula of the verb and the formulas of its arguments. One of the first approaches using a conventional NLP pipeline and a knowledge base was developed

by Hirst in 1987 [79]. His pipeline tokenized the input texts, translated the sentences to a semantic representation and, afterwards, applied a *spreading activation* based approach (i.e., tokens are send around within a knowledge graph, collecting information which identify a structure revealing the sought information) to find the correct meaning of the words. He called his approach "polaroid words", because the result is revealed gradually over several iterations, similar to a polaroid photography [21]. In general, approaches to solving the WSD task can be differentiated into three different categories:

1. Dictionary-based: This approach is based on the observation that the definition of a word within a dictionary requires the use of other words. Those other words, however, also occur in the context of a textual occurrence of the original word. An example is the term "pine cone". Pine has (amongst others) the dictionary definition "kind of evergreen tree with needle-shaped leaves", whereas cone (also amongst others) is described as "fruit of certain evergreen trees". Both definitions share the words "evergreen" and "tree", therefore allowing the identification of the most likely definitions for the single words "pine" and "cone" by searching for the definitions with the highest information overlap [21].

2. Connectionist: Often based upon interest in the way humans thought processes work, connectionist researchers developed corresponding models which have similarities with brain like structures. The approach by Ide and Véronis [80] creates a neural network from a machine readable dictionary, which was constructed of word- and sense nodes. Each word node could be connected to one or more sense nodes. The disambiguation took place by activating selected word nodes which results in tokens spreading the network and increasing the activation of all nodes which they traveled on. In the end, the correct sense nodes for each activated word can be identified as such with the highest activation values. This is similar to the approach by Hirst [79].

3. Statistical and Machine learning: These approaches use large databases to find patterns. An early approach by Brown et al. [81] tried to assign the correct French word to an English word. The problem in machine translation is that an English word may be translated to different French words depending on the context. The system was trained on transcripts of the Canadian parliaments, which are available both in English and French. The concept improved the machine translation results from previously 37% to 45% [21].

The approach taken within the thesis mostly resembles the connectionist approach.

2.1.6.4. Anaphora Resolution

Anaphors are relations between different syntactic entities in a text, which refer to the same real world object. Consider the following example [82]:

> *The queen* is not here yet but *she* is expected to arrive in the next half an hour.

Here the phrase "The queen" is the so called *antecedent*. The word "she" has the role of an *anaphor*, i.e., it points back to the phrase "The queen". Both words also relate to the same real world referent and are therefore called *coreferential* [21]. There are different types of anaphora like *lexical noun phrase anaphors* (i.e., different noun phrases referring to the same antecedent), *verb anaphora* (e.g., "Stephanie *balked*, as *did* Mike" [21]) and many others. For the thesis the most important is the so called *pronominal anaphora* (which is also the most widespread type [21]). In this specific case, a personal, possessive or reflexive pronoun references a specific antecedent. For example, the word "He" could refer to an antecedent noun "Wolf".

The identification of the corresponding antecedents and anaphors is called *anaphora resolution*. The process requires a lot of knowledge. Some systems try to resolve anaphors by using morphosyntactic features, i.e., identifying agreements (see section 2.1.2.2) between the gender, person and numbers between the antecedent and a potential anaphor. Other approaches incorporate knowledge about recency (i.e., words that are further apart are less likely to be anaphoric), grammatical roles (i.e., a subject is more likely to become the antecedent in an anaphoric relation than the object) and verb semantics (i.e., some verbs have semantic implications on their subjects and objects, therefore introducing a "bias" towards the correct antecedent) [20]. Three basic types of algorithms can be differentiated in pronominal anaphora resolution [20]. The *Hobbs* algorithm [83] uses mainly morphological information and a syntactic parse tree to evaluate which pronoun has an anaphoric relation to which antecedent. Another type of algorithms is based on *Centering theory* [84]. In contrast to Hobbs, this algorithm type makes use of an explicit representation of a discourse model (i.e., a model which tries to represent the hearers information of the ongoing discourse [85]) and tries to identify different centers of sentences (i.e., what a sentence is about). It therefore makes use of morphosyntactic information as well as syntactic parses. The third type of algorithm is based on statistical concepts, which are trained on a manually annotated corpus using some of the previously mentioned knowledge types like gender, recency, etc [20].

2.1.6.5. Representation of textual meaning

Representing the meaning of text is always done by introducing *models*, which represent the "state of affairs in the world that we're trying to represent" [20]. Such a model contains

a mapping between the *non-logical vocabulary* of the natural language text and a *logical vocabulary*, i.e., a set of specific operators, symbols and concepts. Creating the mapping is not always straightforward because of all the different types of ambiguities that language can contain (see section 2.1.4 for more information). Hence, such a mapping is also called *interpretation* [20]. In the following we give a short introduction into different approaches to representing textual meaning. A flexible and well-understood approach is *First-Order Logic (FOL)*, which offers among others a good basis for inference and verifiability [20]. Using FOL, a sentence like

All vegetarian restaurants serve vegetarian food.

can be expressed in FOL as follows:

$$\forall x VegetarianRestaurant(x) \Rightarrow Serves(x, VegetarianFood)$$

FOL therefore allows the use of quantifiers, variables as well as functions and predicates to express the statements of certain natural language sentences. The information can be used for inference, i.e., extracting implicit knowledge from given formulas. Another common way of representing the meaning of natural language is *semantic networks*. These are constructed from nodes and links between those nodes which represent the objects and relations between these objects within a sentence. To better understand the semantic meaning of those networks they have often been translated into FOL [20]. This, however, has the drawback that the meaning of such a network depends on the system which interprets it. Therefore, *Description Logic (DL)*, a subset of FOL, has been introduced which eliminates many of those problems. A common way of capturing DL is by using ontologies (see section 2.2 for more information).

2.2. Ontology

The origins of the term *Ontology* date back to the Greek philosopher Plato, who searched for answers to fundamental questions like 'What is reality?' and 'What things can be said to exist?' [2]. The answers to such questions are analyzed in ontology, "the study of what there is" [86]. This field of research therefore cares about all kind of things and entities and how they are related to each other. Although coming from philosophy, ontologies in computer science received a different meaning. Generally speaking "an ontology is a description of knowledge about a domain of interest, the core of which is a machine-processable specification with a formally defined meaning" [2]. A famous definition in this context comes from Gruber [87]:

> An ontology is a (formal) explicit specification of a (shared) conceptualization.

"Conceptualization" refers to an abstract model of some phenomenon in the world for which all the relevant concepts as well as their relations have been identified. "Explicit" means that the type of concepts used and the constraints on those are explicitly defined. "Formal" refers to the fact that an ontology should be machine-readable. "Shared" reflects the notion that an ontology captures consensual knowledge, i.e., it is not the understanding of just one individual but of a group of people. An ontology normally consists of basically four different types of elements [88]. First of all, there are *classes* and *objects*. A class is used within an ontology to create a category of some kind. The category later groups objects. For example a class within an ontology could be a "Computer Manufacturer" (see figure 2.9). An object might be "Dell" or "HP", as both actually manufacture computers. In order to specify structures between classes, *relations* can be used. Relations specify properties which are valid for all objects of this class. In figure 2.9, the relation "manufactures" between "Computer Manufacturer" and "Computer" describes such a relation which is valid for all objects of the category "Computer Manufacturer". One very specific and commonly used relation within ontologies is one which enables creating *taxonomies* (also called *generalization hierarchy* or *inheritance hierarchy*). Taxonomies are used to specify sub- and superclass relations between classes. In our example, such relations are specified by the name "Is A". Therefore, the class "Laptop" is more specific and a subclass of the class "Computer". In order to specify to which class an object belongs, there in this case exists the "A Kind Of" relation. This can be seen between, e.g., "e3" (which is our object representing the "HP Envy Ultrabook") and the class "Laptop". Further, there are relations between objects only. In figure 2.9, both objects "e1" ("Dell") and "e2" ("HP") have a relation "manufactures" on "e3" ("HP Envy Ultrabook") and "e4" ("Dell Vostro"). These relations represent specific properties of the corresponding objects, i.e., "HP" manufactures "HP Envy Ultrabooks", whereas "Dell" manufactures "Dell Vostro" PCs. Relations can further be used to associate concrete values to objects. In our example, each of the ob-

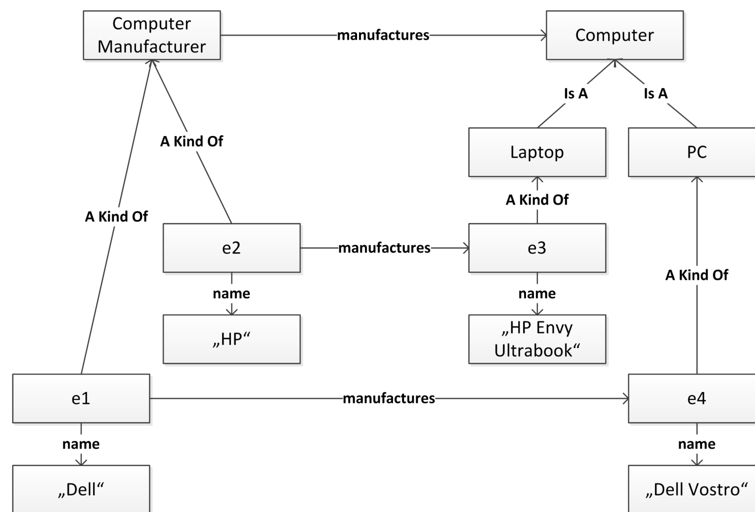


Figure 2.9.: Example of an ontology containing the basic elements

jects has a "name", associating an object like "e1" to a concrete value (here a string with the value "Dell"). These are the basic building blocks of ontologies. Modern languages like OWL (see section 2.2.2) offer more features than the ones which have only briefly been described here. It should, however, be mentioned that within SE-DSNL a lightweight concept of modeling ontologies was developed which offers basically the same features as presented above. Another important aspect of ontologies is if it behaves according to the so called *open world* or *closed world* assumption. The open world assumption specifies that although some information is not contained within the ontology, it does not mean that it does not exist. Hence, it is simply unknown if specific information is valid or not. In contrast, the closed world assumption defines that the ontology contains all relevant information. If information is not contained within the ontology, it simply does not exist and is, therefore, not valid [88].

2.2.1. RDF and RDFS

The Resource Description Framework (RDF) is used for representing structured information, e.g., metadata information about Web resources like the title and author of a web page. This formally well defined structure is suited for applications which want to exchange data [89]. RDF is built upon a graph like structure which is comprised of nodes (called classes) and directed relations (called properties) between those nodes. Both classes and properties are identified by Uniform Resource Identifiers (URIs). If an RDF graph should describe any type of structure, URIs act as unique identifiers. In the context of the internet and web pages, Uniform Resource Locators (URLs) are used which are a specialization of URIs. These allow to de-

Figure 2.10.: An RDF graph describing a person with the name Eric Miller [89]

scribe web page specific structures and information by using RDF graphs. An example of an RDF graph can be seen in figure 2.10. The central class here is identified with the URI "http://www.w3.org/People/EM/contact#me". The element is refined by different properties and values. First of all, there is a property stating that the class is of type "Person" (note that we from here on leave out the prefix "http://www.w3.org/People/EM/contact#"). The property specifies that the element "me" belongs to the category "Person", i.e., it is the RDF way of specifying that an instance belongs to a specific category (compare this to the "A Kind Of" relation from the previous section 2.2). Elements like "me" are also called *individual*, i.e., elements which are a concrete instance of a certain class or category. Next, two literals have been associated with the element "me", i.e., its name "Eric Miller" and his title "Dr.". Further, there is another property "mailbox", which links "me" to an element "mailto:em@w3.org".

In order to serialize RDF graphs, Notation 3 (N3) was developed by Tim Berners-Lee [90]. The standard allows representing RDF graphs as so called *triple*s. A triple consists of a subject, a predicate and an object, whereas the subject presents the source element, the predicate a property and the object the target of a property. An example can be seen in figure 2.11. The figure shows an N3 representation of the RDF graph in figure 2.10. It starts by first specifying the different prefixes used in the graph ("pim", "w3" and "people") and, next, stating the different triples of the graph. As can be seen, the "people:me" element is the subject of every triple because it is the element at the center of figure 2.10. The URIs of the different properties represent the predicates of the triples, e.g., "w3:type" and "pim:fullname". The targets of the different properties finally describe the object of

```
@prefix pim: <http://www.w3.org/2000/10/swap/pim/contact#>
@prefix w3: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix people: <http://www.w3.org/People/EM/contact#>

people:me  w3:type              pim:Person
people:me  pim:fullName         "Eric Miller"
people:me  pim:mailbox          "mailto:em@w3.org"
people:me  pim:personalTitle    "Dr."
```

Figure 2.11.: Representation of the Eric Miller RDF Graph in N3 form

each triple, e.g., "pim:Person" and "Eric Miller".

So far we gave a short introduction into RDF, which provides very basic capabilities to describe the knowledge of a certain domain. The shortcoming of RDF is that it lacks the possibilities of defining deeper semantic relations between elements. In the example in figure 2.10, it is clear to a human that if the individual "me" is a "Person" it necessarily has to have a name and certain characteristics like eyes, legs, etc., all of which is yet missing. It is especially not available to a computer, for which all the information is currently just a set of characters and strings. Therefore, an extension to RDF was developed, called RDF Schema (RDFS). It enables the specification of so called *terminological knowledge* (sometimes also referred to as *schema knowledge*, hence the name RDFS). RDFS "allows for defining a new vocabulary and (at least partially) specifying its 'meaning' in the document without necessitating a modification of the processing software's program logic" [2]. This makes RDFS an *ontology language* because knowledge, encoded with RDFS, is machine readable which matches the initial definition of an ontology in section 2.2. As noted previously, RDFS is an extension of RDF and, therefore, makes use of already existing elements of RDF. To define specific classes and categories RDFS provides the element "rdfs:Class". If someone wants to specify a specific category (e.g., the element "Person" in the previous example 2.10), it can be done as seen in figure 2.12. There, two classes "ex:Lifeform" and "ex:Human" were created which are both of type "rdfs:Class". Further, an "rdfs:subClassOf" property has been introduced between "ex:Human" and "ex:Lifeform", indicating that the class "ex:Human" is more specific than "ex:Lifeform", thereby creating a small taxonomy. Next, an instance of "ex:Human" was introduced, i.e., "ex:me". The individual belongs to the class "ex:Human" and (because of the "rdfs:subClassOf" property) also to "ex:Lifeform".

RDFS provides even more elements which allow a better specification of domain specific knowledge than RDF. An important addition is the better specification of properties, e.g., taxonomies can be defined on properties by using the "rdfs:subPropertyOf" feature. Further, the types of the beginning and end of a user specified property can be restricted

```
ex:Lifeform  rdf:type         rdfs:Class
ex:Human     rdf:type         rdfs:Class
ex:Human     rdfs:subClassOf  ex:Lifeform
ex:me        rdf:type         ex:Human
```

Figure 2.12.: Representation of the Eric Miller RDF Graph in N3 form

by using "rdfs:domain" and "rdfs:range". A complete overview of all additional elements can be found in the RDF Vocabulary Description [91].

2.2.2. OWL

Despite its many features, RDFS has certain problems when it comes to expressivity, e.g., cardinality is a problem ("A car has four wheels"). Expressing such complex information is normally done in formal logic based languages. Formal logics further have the advantage of enabling reasoning and thereby making implicit knowledge available [2]. For those reasons, the Web Ontology Language (OWL) was developed [92] [93] [94] [11]. The design principle behind OWL is to provide "a reasonable balance between expressivity of the language on the one hand, and efficient reasoning, i.e., scalability, on the other hand" [2]. The first official W3C recommendation of OWL arrived in the year 2004. The new version OWL 2 Web Ontology Language (OWL2) was introduced in 2009. In the following we explain the basic mechanisms of OWL. OWL2 is not introduced in detail as it is built upon OWL, and, therefore, only adds some additional features.

To support the balance between expressivity and efficient reasoning, OWL offers different sublanguages, which can be seen in table 2.1. In OWL2 a different approach was taken, i.e., instead of sublanguages profiles have been developed: OWL2 EL enables polynomial time algorithms for reasoning tasks, OWL2 QL provides conjunctive queries and OWL2 RL provides polynomial time rule based reasoning [94]. All these profiles are based upon description logics.

Additionally to the elements of RDFS several new classes and properties have been introduced in OWL. One can create new classes in OWL by using "owl:Class" and assign a name to them with "rdf:about" (note that, in the following, we represent OWL information with the corresponding XML code):

```
<owl:Class rdf:about="Employee"/>
<owl:Class rdf:about="Address"/>
<owl:Class rdf:about="Project"/>
```

The example creates three classes, one with the name "Employee", a second one with the

Sublanguage	Features
OWL Lite	decidable less expressive Worst-case computational complexity: ExpTime Subset of DL and Full
OWL DL	decidable Worst-case computational complexity: NExpTime Subset of Full
OWL Full	Contains all of RDFS Undecidable Very expressive Difficult to understand semantics Hardly any software support

Table 2.1.: OWL Sublanguages [2]

name "Address" and a third one with the name "Project". New instances can be assigned to these classes as follows:

```
<Employee rdf:about="Wolf"/>
<Address rdf:about="UniAugsburg"/>
<Project rdf:about="BRM"/>
<Project rdf:about="ITS"/>
```

Here, a new individual with the name "Wolf" has been created and assigned to the previously created class "Employee". Further, an instance "UniAugsburg" was assigned to the class "Address". Also, two projects "BRM" and "ITS" where created. In order to assign additional information to classes or individuals, properties can be used.

```
<owl:ObjectProperty rdf:about="worksAt">
    <rdfs:domain rdf:resource="Employee">
    <rdfs:range rdf:resource="Address">
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="worksOnProject">
    <rdfs:domain rdf:resource="Employee">
    <rdfs:range rdf:resource="Project">
</owl:ObjectProperty>
```

Here we specified two new properties. The first one states that an "Employee" lives at a specific "Address". Hence, the domain of the property "worksAt" has been restricted to the class "Employee" and the range is limited to the class "Address". The second property "worksOnProject" states that an "Employee" works on a "Project". These new properties can be used to relate the two instances "Wolf" and "UniAugsburg".

```
<Employee rdf:about="Wolf">
  <worksAt rdf:resource="UniAugsburg">
</Human>
```

As can be seen here, the information about the individual "Wolf" has been extended and now contains the property "worksAt", which points to an individual of the class "Address", i.e., "UniAugsburg". We now specify the more complex expression that an Employee has to work on at least two projects. Therefore, the specification of the class "Employee" has to be changed as follows:

```
<owl:Class rdf:about="Employee">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="worksOnProject">
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
            2
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The example specifies that the new class "Employee" has a restriction when it comes to the previously defined ObjectProperty "worksOnProject". If the property should be applied to a class of type "Employee", there must be at least two of these properties such that the minimum cardinality restriction won't be violated. The following example shows how the individual "Wolf" would satisfy this restriction:

```
<Employee rdf:about="Wolf">
  <worksOnProject rdf:resource="BRM">
  <worksOnProject rdf:resource="ITS">
</Human>
```

In the example, the employee "Wolf" is related to two projects "BRM" and "ITS". This is why the previously defined restriction is not violated.
It is important to note that OWL is based upon the open world assumption (see section 2.2). Hence, it is necessary to make the information within an ontology as precise as possible such that the inference mechanisms later does not make any wrong assumptions. One example for this is the "owl:disjointWith" statement which allows to make two classes A and B disjoint. This means that no individual could be assigned to both classes A and B [2]. In contrast, "owl:equivalentClass" specifies that two classes A and B are actually the same, therefore individuals which would be assigned to class A would also be assigned to B. Similar expressions are available to specify that several individuals are ei-

ther different ("owl:AllDifferent") or the same ("owl:sameAs"). Properties can further be detailed by specifying them as being transitive (e.g., if person A is a sibling of person B and B is a sibling of C, then A is also a sibling of person C) or symmetric (e.g., if person A lives near to person B, person B lives also near to person A). These features (among others as shown in [92] [93] [94] [11]) provide ways of modeling semantically complex knowledge which can automatically be validated by reasoners. Further, implicit knowledge can be found and made available to the user or other applications. Despite its impressive logical formalisms, OWL has certain deficits when it comes to linguistic grounding. In the following, we introduce the problem and present an approach which tries to handle the challenge.

2.2.3. Linguistic grounding

One of the biggest challenges with OWL is its linguistic grounding, i.e., how well the semantic information of an ontology can be represented in an arbitrary natural language. OWL itself only allows the definition of one single label for each concept, which more or less acts as a single word representing one concept. Extensions like SKOS allow adding additional labels to single classes [95] [96]. The most elaborate approach to representing ontological information is LexInfo [12], which we introduce in the following. Figure 2.13 gives an abstract overview of the most important elements of LexInfo. At the center, there is the element *LexicalElement*, which has been differentiated into two subclasses *WordForm* and *PredicativeLexicalElement*. The first one represents single verbs, nouns and adjectives, whereas the latter can represent complex predicate-argument structures as well as lexical entries in case of adjectives. Therefore, the element has been further differentiated into subclasses, representing, e.g., transitive and intransitive verbs or scalar and literal adjectives (this can be seen in the original paper [12]). LexicalElements can represent a class (indicated by the "anchor" relation). WordForms in contrast can be used to represent any ontological element.

LexInfo is based on the Lexical Markup Framework (LMF) [97] [98] [99], which "is a metamodel that provides a standardized framework for the creation and use of computational lexicons, allowing interoperability and reusability across applications and tasks" [12]. Two packages of LMF are especially important to LexInfo. Both can be seen in figures 2.14 and 2.15. The first figure presents the structure which allows the definition of the *Syntactic Behaviour* of a *Lexical Entry*. This is done by creating *Subcategorization Frame*s as well as their corresponding *Syntactic Argument*s (e.g., subject, object, predicate, etc.). The type of syntactic information can be associated to its corresponding semantic information by using the elements available within the package shown in figure 2.15. At the core there is the element *SynSemCorrespondence* which defines the mapping between semantic

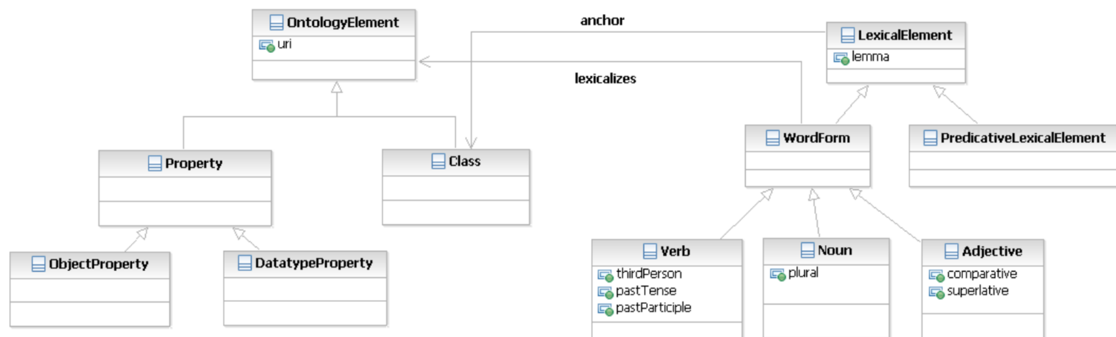


Figure 2.13.: Abstract overview of LexInfo [12]

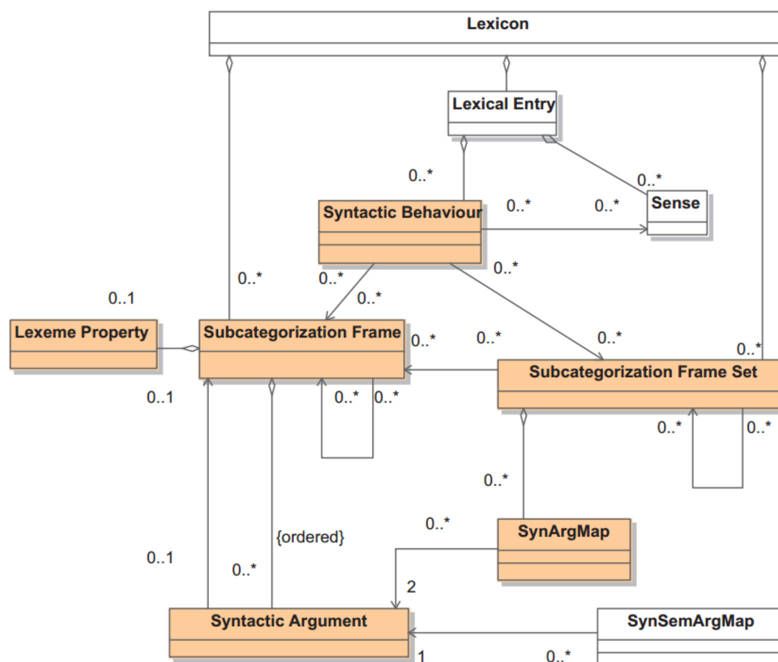


Figure 2.14.: Syntactic extension of Lexical Markup Framework [12]

and syntactic elements. It does so by referencing several *SynSemArgMap* elements which directly associate *Syntactic Argument*s with *Semantic Argument*s of a *Semantic Predicate*. LexInfo especially extends the elements *Predicative Representation*, *Semantic Argument* and *Semantic Predicate* by creating specific subclasses of these elements.

Figure 2.16 shows how this model can be used. There, the lemma "river" is associated with two of its morphological variations. The upper word form is described with the syntactic property value "singular". The word form "rivers" in contrast is normally meant to represent a plural form. A more complex example can be seen in figure 2.17. Here, the term "Autobahnkreuz" (English: motorway intersection) is decomposed into its single components, i.e., "Autobahn" and "Kreuz", which are mapped to their corresponding on-

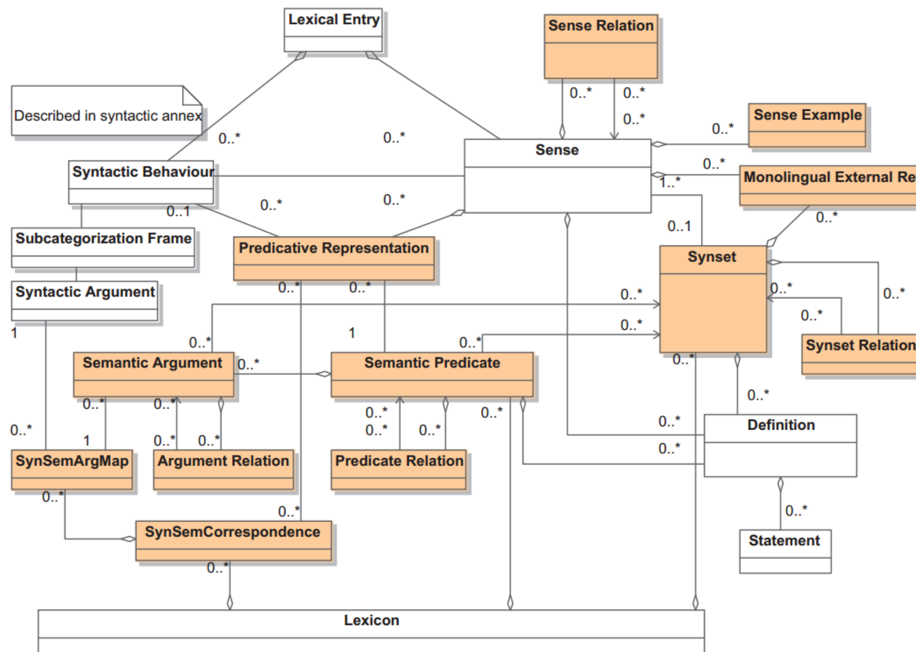


Figure 2.15.: Semantic extension of Lexical Markup Framework [12]

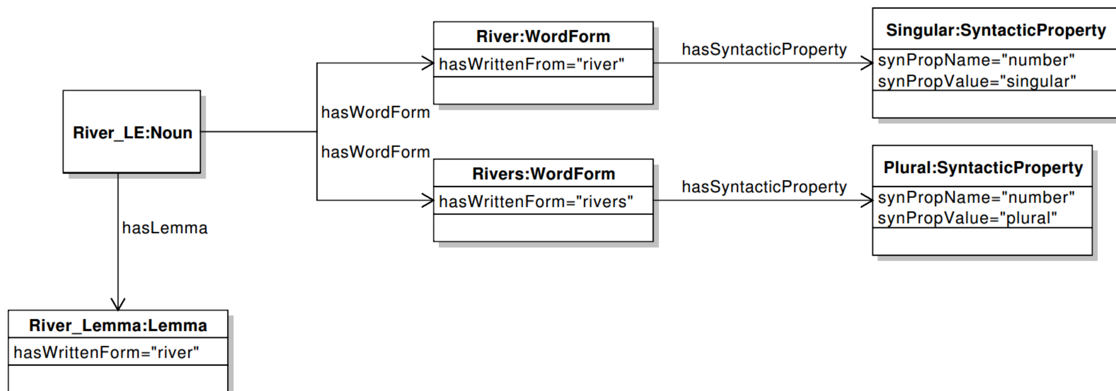


Figure 2.16.: Associating the lemma "river" with some of its morphological variations [12]

tological elements. Starting at the top in figure 2.17, "Autobahnkreuz" is defined as a term which consists of a *ListOfComponents*. The list has two elements: The first component is the one representing the word "Autobahn". An interesting attribute here is "order", which specifies the intended order of the words. The component is associated to an element, specifying that the word is a noun, which has "Autobahn" as its lemma. Further, its sense is "Highway", a class within the ontology. The second component "Kreuz" is specified similarly to the first component. The main difference is that its order value is "2", meaning that "Kreuz" must come after the word "Autobahn".

Another example can be seen in figure 2.18, which shows an example of how the verb

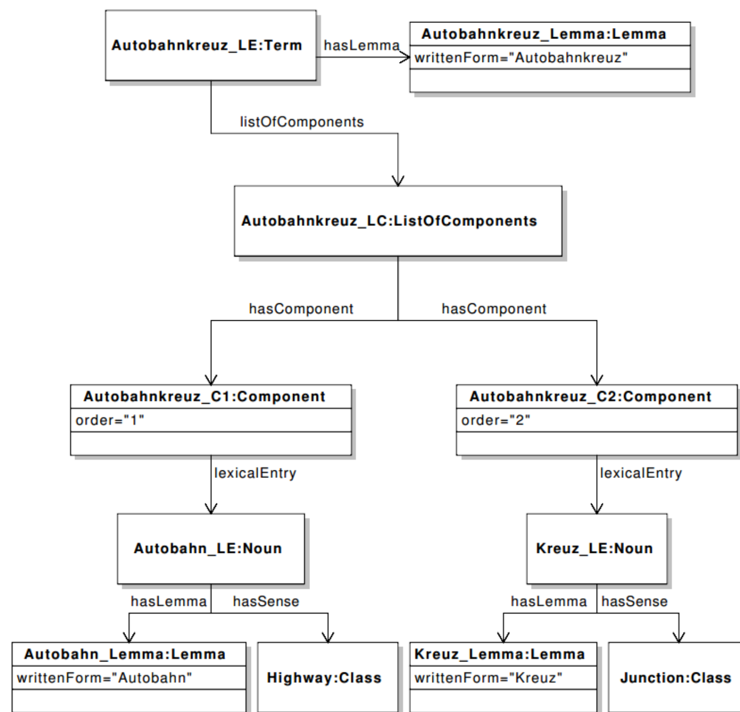


Figure 2.17.: Decomposition of the german word "Autobahnkreuz" into its single components and mapping to ontological elements [12]

"flow" and its arguments are represented within LexInfo. At the core is the element *flow:Verb* (in the lower left of the figure), which represents the verb itself. Again it is represented by a corresponding lemma. For this verb, the syntactic behaviour has been modeled using an intransitive subcategorization frame (*flow_SF:IntransitivePP*). The frame expects two syntactic arguments: A subject and an object. The latter further requires a "through" preposition. On the right side of figure 2.18, the semantic information is represented. At the bottom the corresponding ObjectProperty, which represents the "flow" information within the ontology, is shown. The property and its information are linked to the syntactic information by using the *PropertyPredicativeRepresentation*, which has two arguments: The domain and range of the *flowThrough:ObjectProperty*. The *flowThrough_domain:Domain* element is mapped to the *flow_subject:Subject* element with the *map1:SynSemArgMap* element. The same is done with the object and range element.

The LexInfo approach allows a very detailed mapping between linguistic information (on a lexical and syntactic level) and semantic knowledge. However, it contains no features to represent meaning at a morphological level. A more detailed delimitation between SE-DSNL and LexInfo can be found in section 3.

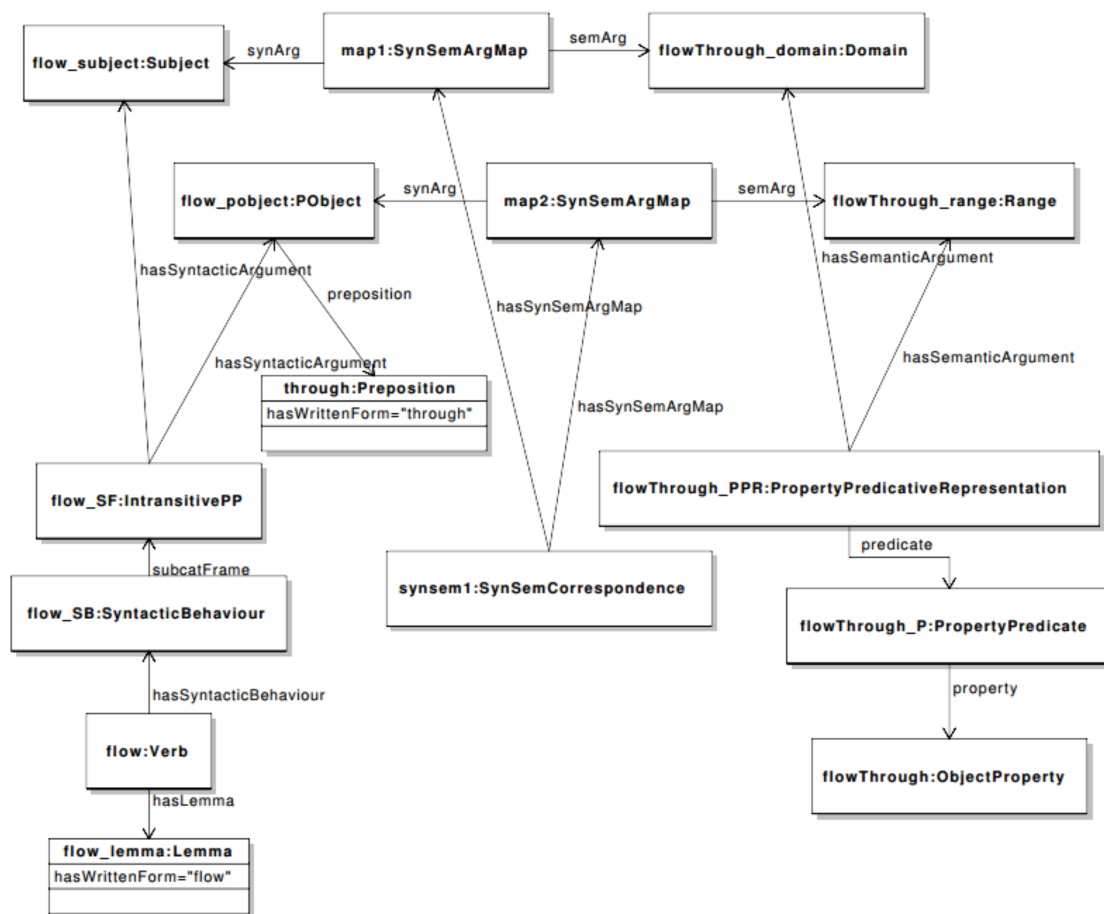


Figure 2.18.: Representation of the syntactic behaviour of "flow" and its mapping to the ontology [12]

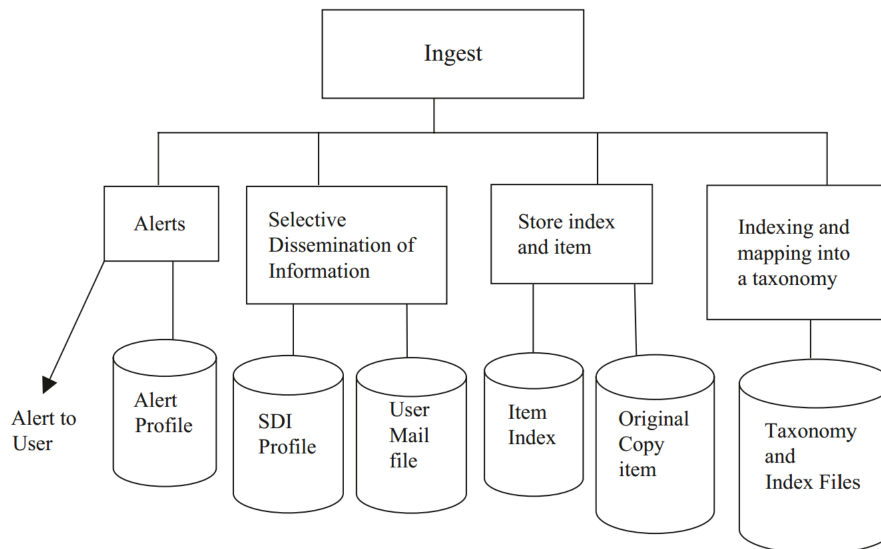


Figure 2.19.: Functional overview of an IR system [101]

2.3. Ontology-based Information Retrieval & Extraction

Today, every person uses web search engines to find specific information. To do so the user enters a set of keywords he / she is looking for (a so called search query). The search engine processes these keywords and returns some elements which match the search query. The process of finding specific information is also known as Information Retrieval (IR) and has been described by Manning et al. [100] as follows:

> Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Most IR systems contain a set of common functions which can be seen in figure 2.19. The *Ingest* function receives new texts and starts processing them. First of all, the new texts are stored in their original raw form and an *index* is created from them (i.e., a searchable data structure based on the original input data). The *Selective Dissemination of Information* "allows users to specify search statements of interest (called 'Profiles')" [101] and retrieves texts which match those search statements. If a new data item has been found, the user can receive an *alert*. The final functionality *Indexing and mapping into a taxonomy* may assign metadata to the items and store them in a taxonomy. The taxonomy can be navigated by users to find other elements of interest.

The indexing and metadata assignment tasks incorporate the challenge of identifying information which are relevant. If every information would be added to the index, the corresponding database would contain unnecessary information and the overall precision

would suffer. The task of identifying relevant information and extracting them is also called Information Extraction (IE) [101]. Cowie and Wilks [102] defined IE as follows:

> Information Extraction (IE) is the name given to any process which selectively structures and combines data which is found, explicitly stated or implied in one or more texts.

Depending on the level of detail and precision required, different information is extracted from a text. Standard, syntactic based search engines rely on indexing nearly every word which is not a stopword (i.e., words without any relevant semantic content, e.g., "the", "a", punctuation marks, etc.). In contrast, some approaches try to only extract information which are relevant to a specific task. For example the challenge of the Message Understanding Conference 4 was to extract information about terrorism from different newspapers [103]. The extracted information consisted of information about which country had been attacked, which weapons had been used etc.. Systems which deal with IE only, have to face similar problems like those which deal with IR. The reason is that both types have to deal with the challenges of analyzing and parsing unstructured natural language text and the corresponding linguistic phenomena like homonymy, synonymy, anaphoras etc. (see section 2.1). Therefore, we often mention both types of systems in the following because of similar problems they have to cope with. In order to overcome those problems standard IR and IE approaches have been extended with ontologies in the last couple of years, leading to so-called *Ontology-based Information Retrieval (OBIR)* and *Ontology-based Information Extraction (OBIE)* systems. Here, ontologies provide a source of semantic knowledge which can be used in different ways to enhance the overall precision of those systems. Wimalasuriya and Dou [104] gave a definition of what best describes an OBIE system:

> An ontology-based information extraction system: a system that processes unstructured or semi-structured natural language text through a mechanism guided by ontologies to extract certain types of information and presents the output using ontologies.

It is obvious that this definition and the one of regular IE are very similar. The important part is that in contrast to regular IE systems an OBIE is guided by an ontology. Such an ontology can help identifying the semantic types of specific words as well as possible relations between words. Wimalasuriya and Dou presented different characteristics which are specific to OBIE systems:

1. Processing (semi-)unstructured natural language text: An OBIE system must work on standard natural language documents employing NLP techniques. Systems, which use ontologies to extract information from images, diagrams or videos can not be characterized as OBIE systems.

2. Making use of ontologies to present the output: The most important feature of an OBIE system is to represent the output using an ontology. Some systems also use an ontology as an input, however, Wimalasuriya and Dou think that this is not a necessary requirement.

3. Using the ontology during the IE process: Existing IE systems are often extended by using an ontology which helps during the extraction of classes, instances and properties. Some authors further argue that the *information extractors* should be considered as a part of the ontology itself [105] [106] [107] [108], thereby enhancing the combination of language and semantic knowledge.

Like OBIE, OBIR shares many of those characteristics. The main difference is that OBIR systems require an additional component which retrieves documents matching the search query of a user.

The SE-DSNL approach can be classified as an OBIR system. It parses unstructured natural language text, creates an ontology from it (called an InterpretationModel, see section 3.3.5) and also uses an existing ontology during the extraction process for validation as well as identification of new knowledge. It further provides a component to retrieve information from InterpretationModels which match certain criteria (see section 6). It should be noted that the retrieval process in SE-DSNL is two folded: First it allows the retrieval of texts which match certain semantic conditions. Further, it provides a possibility to retrieve specific semantic information from texts.

A generic architecture of an OBIE system, according to Wimalasuriya and Dou [104], can be seen in figure 2.20. The figure presents an overview of commonly used components. First, an expert normally creates an ontology and provides a set of components which are responsible for the extraction of required information. These components sometimes make use of the ontology as well as other sources like a semantic lexicon (e.g., WordNet [13] [14]) to validate the information which they extract. Analyzing an input text normally consists of several preprocessing steps (e.g., splitting the text into its single sentences and words, removing stopwords, creating a syntactic parse tree, etc.). After the preprocessing, the IE modules extract information and put the results into a knowledge base. In some systems these results are not written into an additional knowledge base, but directly into the ontology. SE-DSNL also matches this architecture: An expert has to create the ontology as well as the IE modules (called Constructions, Statements and Functions as described in section 3.3.4, 4.3 and others). Text is preprocessed and the results are written in an additional knowledge base (the interpretation models). The difference between SE-DSNL and other approaches is the much deeper integration of linguistic knowledge and how the information of the ontology can be used during the analysis process. All of this is presented in the following chapters.

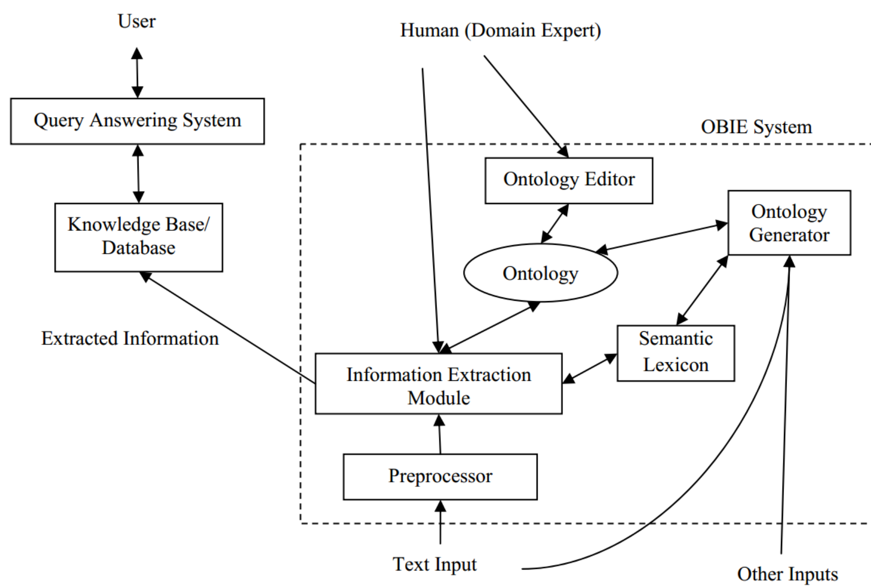


Figure 2.20.: General architecture of an OBIE system [104]

3

Combining Ontological and Linguistic Information

3.1. Introduction

Natural language is the most common way for humans to communicate. The communication itself can happen in different ways, i.e., humans can speak to each other or write their thoughts down, e.g., on a sheet of paper. In this thesis, we completely focus on written natural language or, simply, text. On the surface, text consists of single words, each of which is constructed from a set of letters, numbers, etc.. Each word normally has a specific meaning. However, due to ambiguity, it is often difficult to identify the meaning as it depends on the context of the words itself. Next, each language has its own way of expressing meaning in a text, i.e., depending on the morphology of the words and the syntax of the language the author can change the meaning of the text as a whole. Also, different languages exhibit different ways of creating words and sentences. This is easy to see in the words themselves, i.e., different languages often have very different words for expressing the same meaning (e.g., the English word 'car' and is represented in German by the word 'Auto') and also different syntactical rules. Whereas in German and English the order of the words is very important to the meaning of the text, Czech in contrast allows a more flexible approach to its word order (for more information have a look at section 2.1). However, one thing that nearly all people, who speak different languages, have in common, is that they are capable of learning other languages and therefore can understand what others are saying. This basically means that every person which speaks two or more languages, is capable of understanding and expressing his / her knowledge in multiple ways. Another way to put this is that people have only this one knowledge, they do not need to store each knowledge fact / experience / etc. separately for every language they learn.

Handling and representing knowledge in computers is difficult. A prominent way of storing knowledge is represented by ontologies. They provide mechanisms and standards to store facts about specific domains. One of the most important features about ontologies is that they are formal, i.e., they have been designed such that computers can use the information within the ontology to create new knowledge (also known as 'reasoning'). However, ontological concepts have rarely been developed such that they are easy to integrate with human related tasks. Information retrieval can especially benefit from the integration of semantic knowledge, as mentioned in the previous section 2.3. This, however, requires a better mapping between ontologies and linguistic knowledge. In this chapter we describe the SE-DSNL meta model which has especially been designed to contain a mapping between ontological and linguistic knowledge.

The chapter is structured as follows. Section 3.2 first introduces a set of requirements which have to be fulfilled in order to completely map linguistic to ontological knowledge. Next, we present the SE-DSNL meta model in section 3.3 and its different scopes. In

section 3.4 we specify a formal definition of the meta model which is used throughout the rest of the thesis. Following, section 3.5 describes how knowledge from existing OWL ontologies can be transformed to the SE-DSNL meta model. It also describes a set of guidelines which a valid SE-DSNL model must conform to. Finally, section 3.6 delimits our approach from others.

3.2. Requirements

Humans capabilities of analyzing meaning while communicating is unmatched by any available computer component. Although state-of-the-art NLP components are sophisticated regarding their syntactic parsing features, semantic parsing capabilities are not available on a broader scale. Especially domains which already posses semantic knowledge face challenges when trying to use the knowledge in linguistically motivated scenarios. Still, semantic knowledge can provide many benefits to computational linguistics (e.g., by resolving homonyms and ambiguities in general as well as resolving anaphoras as described in section 2.1.4). The reason, why knowledge within domains is yet difficult to use in NLP related tasks is that the concepts and standards are missing to provide a bridge between linguistic and semantic knowledge. We first talk about multilingualism, before we introduce a set of requirements which are necessary in order to fully annotate semantic knowledge in a way that it can be used in linguistically motivated scenarios.

In our modern world, companies span the globe, operating on different continents in different cultures. Companies have to use their knowledge in different countries. This means, they have to express it in different languages. Inhabitants of countries, which are geographically close, use languages which share many commonalities. An example is French, German and English which contain similar syntactic structures and for some concepts even share the same words (e.g., the German and English word "Kindergarten"). However, other languages have developed different ways of encoding information, both on a morphological and syntactic level ([3]):

1. Languages like Vietnamese and English which especially make use of the word order to specify the participants within a sentence.
2. Languages like Hebrew which specify the participants of a sentence by putting a preposition in front of it. In contrast, Japanese uses postpositions to mark the roles of the relevant nouns.
3. Russian and Latin change the endings of the nouns themselves in order to mark them as being a subject or an object.
4. A fourth language type (e.g., dialects of modern Aramaic) does not use additional words or change the nouns ending, but instead changes the ending of the verb to indicate the roles within a sentence.

Except for the first type of languages (English, Vietnamese, etc.) all other language types are more or less independent of word order, as the roles of the nouns are encoded differently. Further, languages like Latin or Semitic languages like Arabic, Aramaic and Hebrew have developed complicated ways of encoding information about time, num-

bers and places within the verbs they use. These differences have to be considered if a standard for mapping semantic to linguistic knowledge should be capable of expressing every available language.

In [108], Buitelaar et al. described five requirements which have to be fulfilled to linguistically ground an ontology:

1. Capture morphological relations between terms, e.g., through inflection (cat, cats), separately from the domain ontology,
2. Represent the morphological or syntactic decomposition of composite terms and the linking of the components to the ontology,
3. Model complex linguistic patterns, such as subcategorization frames for specific verbs together with their mapping to arbitrary ontological structures,
4. Specify the meaning of linguistic constructions with respect to an arbitrary (domain) ontology,
5. Clearly separate the linguistic and semantic (ontological) representation levels.

We definitely agree with the first four requirements. However, we want to point out that there are some things that should be considered regarding the fifth requirement.

In the human mind, both semantic and linguistic knowledge are interconnected (which has also been proven by different experiments [109], [110]). This means that humans are incapable of clearly separating semantic from linguistic information because language is required to describe (semantic) knowledge. The problem can already be seen when ontologies should be created. In order to identify an ontological element a group of people assigns a name or id to each ontological element, most likely in a readable form. This actually means that in order to represent semantic knowledge a certain linguistic layer is needed to sufficiently and user-friendly manage and retrieve the knowledge at later times. However, the word which is chosen for an element depends on the people within the group and the way they are used to describe the knowledge to their surrounding people. Another group of people which has no connection to the prior group, might for example have very different terms to express the same ontological knowledge. This means that ontologies from different groups, which describe the same knowledge, can most probably not be matched directly because of different terms which have been used to represent ontological elements. Therefore, requirement 5 is desirable from a technical point of view (e.g., to more efficiently implement the data structures or to easily exchange linguistic information between different ontologies or applications). However, from a cognitive point of view, semantic and linguistic knowledge should be closely related and treated with the same priority.

Besides the five previous requirements, we add two additional requirements since the prior ones do not allow to cover certain linguistic information which might be relevant for certain scenarios like NLP. The first one is the grammatical representation, especially in different languages, i.e., certain (sets of) concepts are represented by more complex linguistic structures (e.g., proverbs). These can not be covered by a direct one to one mapping of words to concepts or by subcategorization frames. Instead, the context of the words and their syntactic relations have to be captured to resolve the correct meaning. Hence, e.g., complete phrase- and sentence structures have to be captured and mapped by the linguistic model.
The second requirement deals with the fact that different languages contain different grammatical structures and also different syntactic categories (this has been described in section 2.1). The information can be very helpful to match the linguistic information to the result of existing NLP components. Therefore, the meta model must be easily extensible to cover an arbitrary number of syntactic categories. These further have to be combinable in arbitrary ways. This allows the definition of complex syntactic features, e.g., adding the gender of a noun or the tense of a verb. The new requirements can be summarized as follows:

6. Support for mapping complex grammatical structures allowing, e.g., the representation of proverbs. These more complex grammatical structures have to be related to their corresponding concepts.
7. Support for an arbitrary number of syntactic categories which can be combined in arbitrary ways.

In the following, we show how the meta model of SE-DSNL fulfills all these requirements. Our approach is delimited from similar concepts in section 3.6.

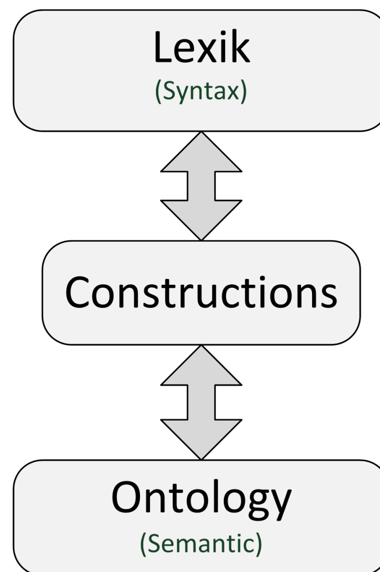


Figure 3.1.: Overall approach for the design of the meta model

3.3. SE-DSNL Meta Model

As stated in section 1.3.1, it is one objective of SE-DSNL to map ontological to linguistic information. The approach we use is shown in figure 3.1. At the top is the lexical part, i.e., it holds all the forms which can be used to represent ontological elements. Further, it contains the syntactic categories, which represent certain grammatical functions. At the bottom of the figure, there is the element "Ontology", which contains the semantic information only. In order to bridge these two, the "Constructions" layer in the middle contains the elements which for one map the single forms to their corresponding ontological element, and are further capable of describing syntactic structures and how these can be mapped to the ontological structures.

Creating a consistent metamodel which allows to fulfill all the requirements specified in section 3.2 can be done in two different ways. One would be to create a very complex and large meta model which can store all information necessary (the approach has been followed by LexInfo). The other way (which was taken here) is a generic and smaller approach which requires more work in the implementation phase, but is more flexibel towards later changes. The following meta model fulfills the previous requirements. In the following lines, the meta model, the considerations which led to it and why they fulfill the requirements is presented.

Figure 3.2 shows an overview of the meta model structure. There, basically, are five different parts which are specializations of the element *Scope*, which defines the commonly used elements. The main element and also the container element is the *Domain* class.

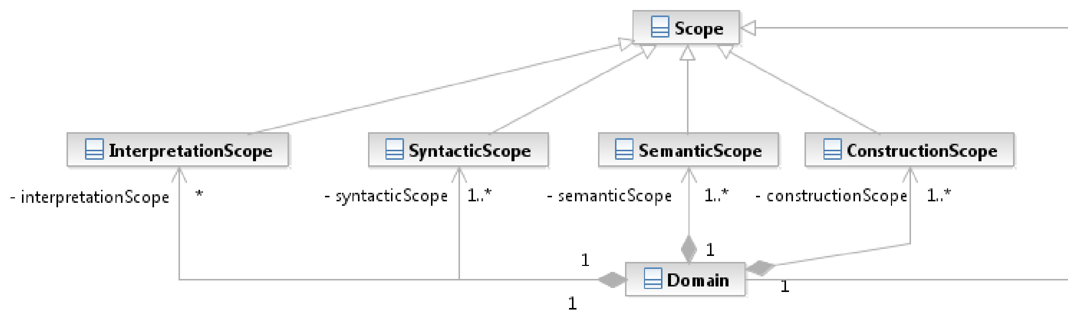


Figure 3.2.: Overview of the meta model structure

It contains at least three different scopes: The *SemanticScope*, the *SyntacticScope* and the *ConstructionScope*. The SemanticScope holds the semantic knowledge of the corresponding domain. This can be compared to the information which are contained in an OWL ontology. The SyntacticScope contains information about the forms as well as syntactic categories. The last mandatory scope is the ConstructionScope. In this one the links between the SemanticScope and the SyntacticScope are defined, i.e., how the syntactic information can be mapped to the SemanticScope. The final scope is the *InterpretationScope*. It is not filled by any expert but instead holds the automatically generated mappings between a natural language text and the SemanticScope of this Domain.
A detailed introduction to the different scopes is given in the following sections.

3.3.1. Scope

One technical consideration for the following meta model was that it is consistent in its approach, i.e., semantic and syntactic information are both contained within their specific scopes, but can still be linked to the other scopes consistently. Hence, the meta model needs a generic upper part, which provides all the functionality that is required by all the other scopes and elements.
The model can be seen in figure 3.3. The containment element is the *Domain* class. As the name implies the class contains everything that needs to be represented within a specific domain. To keep the amount of elements which are directly contained within the Domain as small as possible, a single upper element was chosen, the *ReferencableElement*. A ReferencableElement is an element, which can reference itself, but can also be referenced by other ReferencableElements. Every other element which is introduced from hereon inherits from ReferencableElement. This also implies that every other element can be referenced. The first two new elements for which this is true, are the elements *Relationship* and *Element*. These two classes build the basis for generic graph like structures. Each

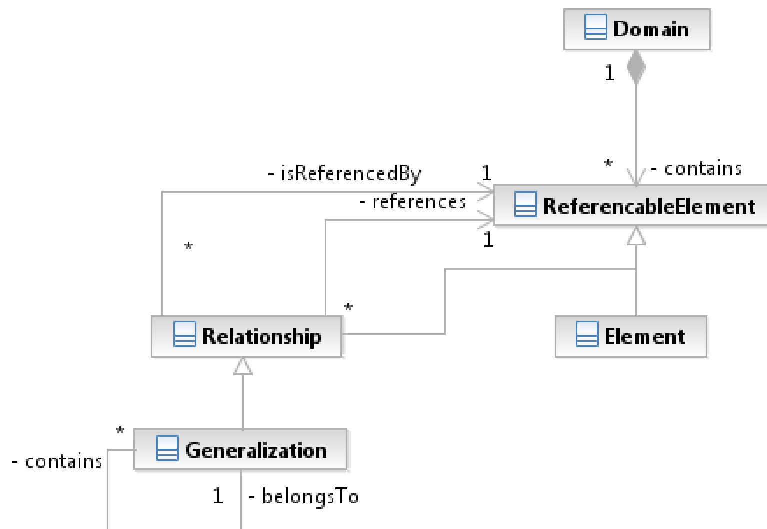


Figure 3.3.: Content of the default Scope of the meta model

Element can have one or more Relationship onto any ReferencableElements and, therefore, also on other Elements or Relationships. The advantage of the approach can be seen in contrast to existing standards for knowledge representation (e.g., RDF or RDFS), which sometimes have difficulties representing statements like "The detective assumes that the gardner killed the man". To represent the fact it would be necessary to reference the verb "kill". Verbs are most likely expressed as properties within an ontology. However, properties can not be referenced directly within RDF or RDFS. In contrast to these approaches, SE-DSNL allows referencing Relationships, which means that reification is not necessary. The previous example can therefore be represented as shown in figure 3.4. There, solid lines represent relations / properties and dashed lines point to the element which indicates the type of a relation (this is explained in more detail in the following section).

A final decision for the scope was that it contains the *Generalization*, which is a specialization of the Relationship. It basically indicates that one ReferencableElement is more specific than another one, i.e., $Gen(a,b) := a \subseteq b$, i.e., for two *ReferencableElement*s a and b, a is equal to or more specific than b. The default Scope builds the foundation for all remaining scopes.

3.3.2. Semantic Scope

Until now only the part of the meta model which builds the common foundation has been specified. Figure 3.5 depicts the part of the meta model which contains the semantic

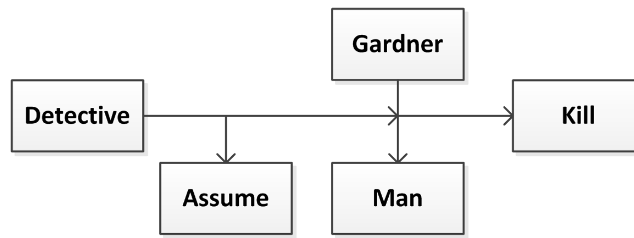


Figure 3.4.: Representation of the sentence "The detective assumes that the gardner killed the man"

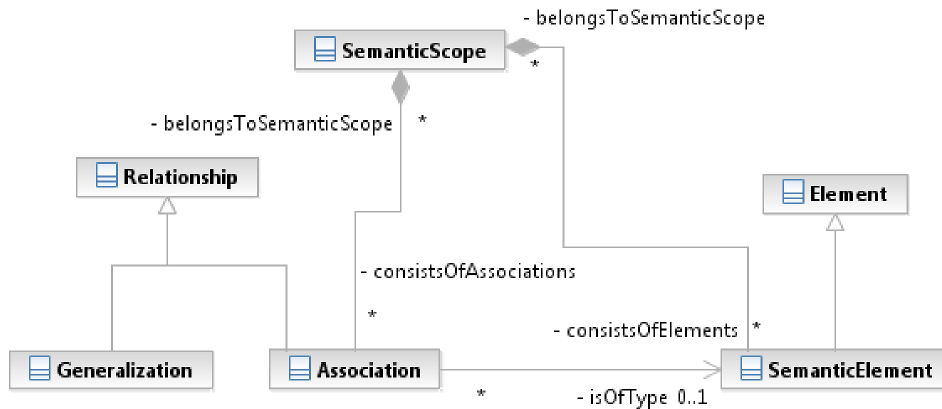


Figure 3.5.: Semantic Scope

information. There are basically two more elements: The *SemanticElement* (which inherits from Element) is used to model each concept which is relevant to a domain. SemanticElements can be related to each other with an *Association*. An Association is a Relationship, however, it has one additional feature: Its type is specified by referencing a *SemanticElement*. Basically, the type of an Association is not specified by the Association itself (in contrast to other standards, e.g., ObjectProperties in OWL). Instead, a link to an external element is used. The implication of the approach is that everything which should be modeled must be created as a SemanticElement, be it an actual process or a static object. An example is shown in figure 3.6. As can be seen, *Person*, *House* and *liveIn* have all been represented as SemanticElements. An Association connects Human and House and specifies its type by referencing liveIn. The advantage of this idea is that every kind of semantic information has to be mapped to linguistic information just once, as one SemanticElement could be referenced by an arbitrary number of Associations (this is shown later on). There is also no separation between conceptual and instance level. Basically everything can be a concept or an instance. The decision has been made because of two reasons:

1. A separation between instance and concept level would make more sense if there

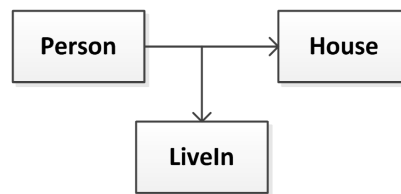


Figure 3.6.: Example of the structure for "Person lives in House"

would be an accurate way to identify which level a word in the text actually refers to. However, it is often difficult to know if a word like, e.g., "Car" actually refers to the concept "Car" or, based on its context within a text, to an actual instance. As there is no known algorithm to identify whether a word should be mapped to a concept or a specific instance, the problem was neglected.

2. SE-DSNL does not provide any type of logical reasoning. Further, it is not yet intended for large amounts of data. Instead, SE-DSNL is intended to give a first prototypical view at providing a powerful bridge between the world of semantics and language.

3.3.3. Syntactic Scope

In this subsection we explain which type of syntactic information can be entered. An overview of the scope is shown in figure 3.7. The central element is the *SyntacticElement*. There are basically three specializations: *Form*, *FormRoot* and *SyntacticCategory*. The FormRoot is used to define the root of a word, i.e., the string representation of the root of a word. The Form element specifies any inflections of a word, more precisely any other strings which represent a specific word. FormRoots can be linked to a group of Forms. These Forms should be different inflections of the specific FormRoot. The third element of the Syntactic Scope is the SyntacticCagetory. It can be used to represent any type of syntactic information like, e.g., the different phrasal as well as lexical categories which can be inherent to a specific language.

3.3.4. Construction Scope

Until now we have shown how we intend to model semantic as well as syntactic information. In this subsection we specify how these two parts can be combined. As it was mentioned earlier the basic idea to join semantic and syntactic information comes from Construction Grammars, i.e., a set of Construction each of which consists of a semantic and a syntactic part. This basically is what the element *Construction* represents (see figure 3.8). Inheriting from *ConstructionElement*, the central element of the Construc-

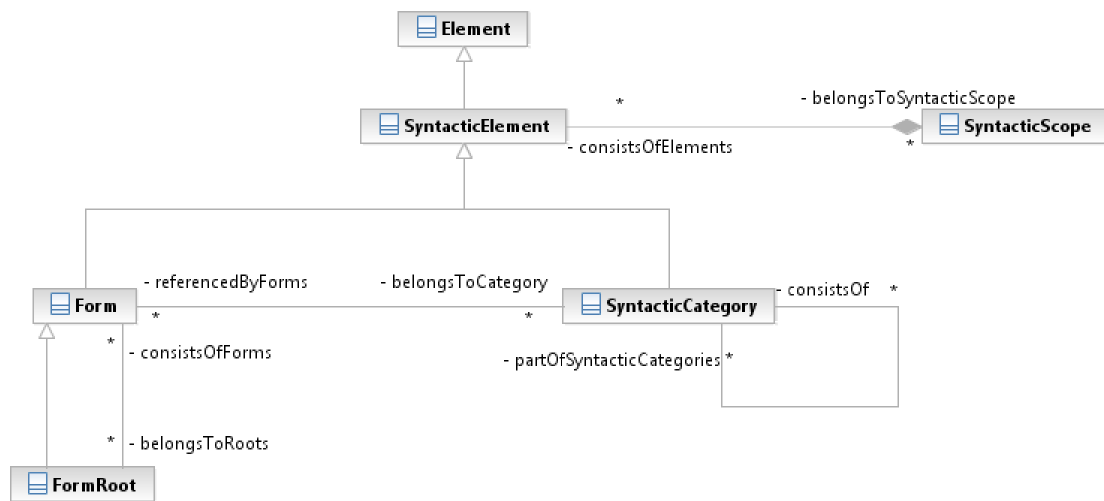


Figure 3.7.: Syntactic Scope

tion Scope, it consists of a set of *Symbol*s as well as *Statement*s. A Symbol can be used to reference any type of information, be it SemanticElements (*SemanticSymbol*), SyntacticElements (*SyntacticSymbol*) or other Constructions (*ConstructionSymbol*). Referencing other Constructions means that information which has been built by other Constructions can be reused. We make extensive use of the information later in section 4.2. In order to analyze natural language text, knowledge about syntactic structures and their mapping to semantic structures has to be encoded into a Construction. Each Construction therefore can specify a set of *Statement*s. A Statement defines a *Function* which receives a list of Symbols as its arguments. An example would be a Statement which checks a list of forms for a specific order in which they should appear in a text. Detailed examples of Constructions are shown and described, e.g., in figure 3.11 of section 3.5.3.

3.3.5. Interpretation Scope

The value of the previous scopes is to contain the information which can be used during the actual analysis. In the process we create what we call a semantic interpretation of the textual input. To save the interpretation a separate scope has been created which allows the storage of different interpretation models and how their elements are connected to the Semantic-, Syntactic- and ConstructionScope (see figure 3.9). As the analysis component can receive an arbitrary amount of requests, there must be a way to store an independent number of those requests. This is supported by the element *InterpretationScope*, which may contain an arbitrary amount of *InterpretationModel*s. This in turn references an arbitrary number of *InterpretationElement*s, which is the default element of the scope. Similar to the default scope an *InterpretationRelation* is an InterpretationElement itself and

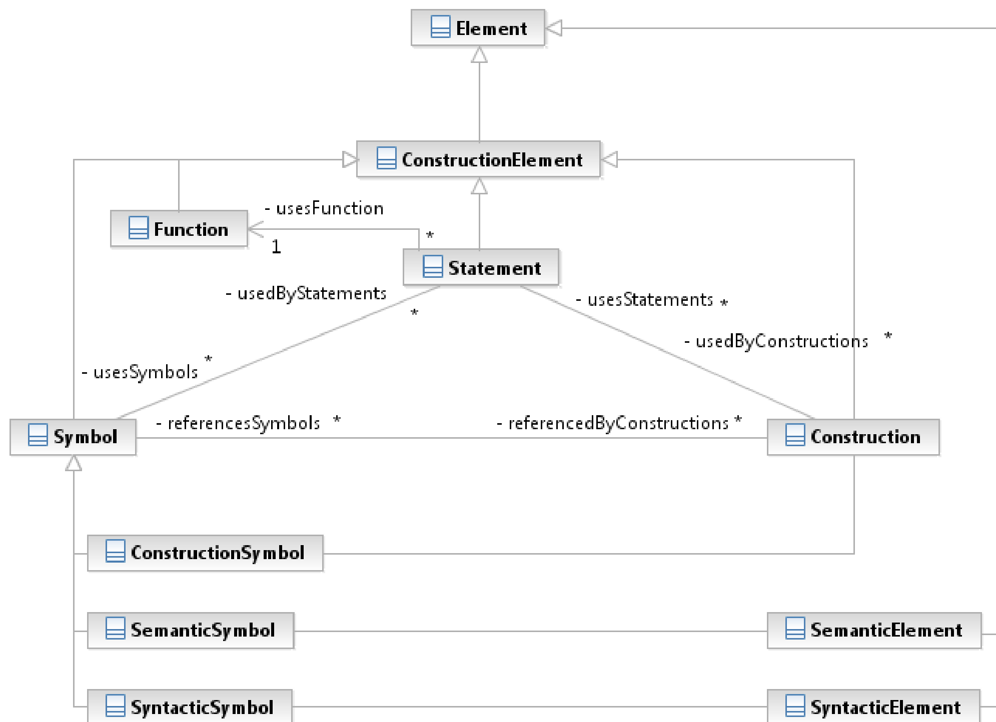


Figure 3.8.: Construction Scope

references different InterpretationElements. The structural similarity had to be repeated as the information of the parsing process could theoretically posses the same structures as those of the SemanticScope. During the parsing process, information from all three different prior Scopes can be identified. In order to clearly identify which information the analysis process is actually referring to, there are three different specializations of the InterpretationElement:

1. The *SemanticElementInterpretation* is, as the name implies, an element which acts as an interpretation for a specific SemanticElement. The SemanticElement must be part of the SemanticScope of the meta model.

2. The *SyntacticElementInterpretation* acts as an interpretation for a SyntacticElement. The SyntacticElement must (of course) also be a part of the SyntacticScope of the meta model.

3. The probably most important element is the *ConstructionInterpretation*. A ConstructionInterpretation represents the persistent version of a Construction instance (as described later in section 4.2.2). It references the Construction which has been used to create the specific instance of a ConstructionInterpretation. Further it references

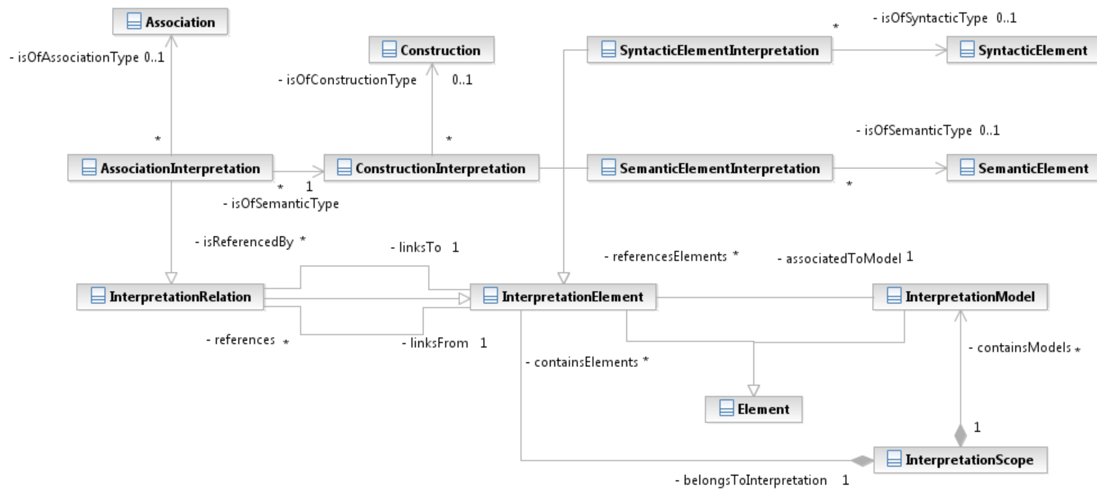


Figure 3.9.: Interpretation Scope

those elements, which have either been newly created (e.g., a new *AssociationInterpretation a* between two other ConstructionInterpretations ci_1, ci_2) or are relevant for the new ConstructionInterpretation (e.g., the two ConstructionInterpretations ci_1 and ci_2). A ConstructionInterpretation is allowed to only reference one single SemanticElementInterpretation (because the semantics of a ConstructionInterpretation must be unambiguous).

In order to relate ConstructionInterpretations, there is one more specific relation available, the *AssociationInterpretation*. This is different from the SemanticScope, where an Association can only be used to associate different SemanticElements. The reason for associating ConstructionInterpretations is that the parsing process needs many different information in order to create an InterpretationModel. The core elements of the parsing process are always the Constructions, as they contain and link all the relevant semantic and syntactic information. This also means that each ConstructionInterpretation (as it is an instance of a Construction) represents its own semantics by referencing a SemanticElementInterpretation. Therefore, instead of directly relating the different SemanticElementInterpretations, the ConstructionInterpretations are associated by using the AssociationInterpretation. Similar to a normal Association, an AssociationInterpretation must also specify the actual type of the relation, which is also a ConstructionInterpretation.

3.4. Formal Specification

During the thesis we often have to refer to different elements of the meta model. We therefore introduce the meta model in a more formal representation. The formalization is not intended to be applied in mathematical proofs, but for the clear specification of algorithms and concepts.

The basis of this concept builds on an instance of a SE-DSNL meta model, which is sometimes called SE-DSNL model or just ontology. It is defined as follows:

Definition 1 (*SE-DSNL Ontology*)

A SE-DSNL Ontology O (also referred to as ontology only) is defined as the set

$$O := \{O_{se}, O_{sy}, O_c, O_i, O_p\} \tag{3.1}$$

where O_{se} represents the SemanticScope, O_{sy} represents the SyntacticScope, O_c represents the ConstructionScope, O_i represents the InterpretationScope and O_p represents the PatternScope.

The definition of the SemanticScope of an ontology O is given in the following:

Definition 2 (*SemanticScope*)

$$O_{se} := \{O_{se}.E, O_{se}.R, O_{se}.G\} \tag{3.2}$$

where $O_{se}.E$ is a set of SemanticElements, $O_{se}.R$ defines a set of Associations between the SemanticElements in $O_{se}.E$ and $O_{se}.G$ defines a set of Generalizations between the SemanticElements in $O_{se}.E$. Each Association $r \in O_{se}.R$ is defined by the set

$$r := \{r.src, r.trg, r.typ\} \tag{3.3}$$

where $r.src \in O_{se}.E$ defines the source SemanticElement of a Relationship, $r.trg \in O_{se}.E$ specifies the target SemanticElement and $r.trg \in O_{se}.E$ specifies the SemanticElement which represents the type of the Association. All Generalizations in $O_{se}.G$ are defined accordingly with the exception that they do not contain a type attribute.

The definition of the SyntacticScope can be seen in the following:

Definition 3 (*SyntacticScope*)

The SyntacticScope O_{sy} consists of

$$O_{sy} := \{O_{sy}.F, O_{sy}.X\} \tag{3.4}$$

where $O_{sy}.F$ is the set of all Forms and FormRoots and $O_{sy}.X$ contains the Syntactic-Categories.

The ConstructionScope is formally defined as follows:

Definition 4 (*ConstructionScope*)

The ConstructionScope consists of the set

$$O_c := \{O_c.C, O_c.Y, O_c.S\} \tag{3.5}$$

where $O_c.C$ is the set of Constructions, $O_c.Y$ is a set of Symbols and $O_c.S$ is a set of Statements which can be used by the Constructions within $O_c.C$. $O_c.Y$ is defined as

$$O_c.Y := O_c.Y_c \cup O_c.Y_{se} \cup O_c.Y_{sy} \tag{3.6}$$

where $O_c.Y_c$ is the set of all ConstructionSymbols, $O_c.Y_{se}$ is the set of all Semantic-Symbols and $O_c.Y_{se}$ is the set of all SyntacticSymbols. $O_c.S$ consists of the following sets

$$O_c.S := O_c.S_e \cup O_c.S_c \tag{3.7}$$

where $O_c.S_e$ is the set of all ConditionStatements and $O_c.S_c$ is the set of all EffectStatements.

In order to map the elements from O_{se} to O_{sy}, Constructions are used. A Construction is defined as follows:

Definition 5 (*Construction Definition*)

A Construction $c \in O_c$ is defined as

$$c := \{c.Y, c.S_C, c.S_E\} \tag{3.8}$$

where

$$c.Y \subseteq O_c.Y \tag{3.9a}$$

$$c.S_C \subseteq O_c.S_e \tag{3.9b}$$

$$c.S_E \subseteq O_c.S_c \tag{3.9c}$$

We use different methods and predicates to return specific values or check certain values,

which can be seen in definition 6.

Definition 6 (*Methods and Predicates*)

In order to return the set of all Symbols which are used by a specific set of Statements, the following method is used:

$$t \subseteq O_c.S \tag{3.10a}$$

$$symb(t) \subseteq O_c.Y \tag{3.10b}$$

The set of Symbols $O_c.Y$ can contain different types of Symbols like Construction-Symbols, SyntacticSymbols and SemanticSymbols. In order to identify the type of a Symbol, new predicates are introduced:

$$conSymb(y) \Leftrightarrow y \subseteq O_c.Y_c \tag{3.11a}$$

$$synSymb(y) \Leftrightarrow y \subseteq O_c.Y_{sy} \tag{3.11b}$$

$$semSymb(y) \Leftrightarrow y \subseteq O_c.Y_{se} \tag{3.11c}$$

where $conSymb(y)$ checks if y is a ConstructionSymbol, $synSymb(y)$ validates if y might be a SyntacticSymbol and $semSymb(y)$ checks if y is a SemanticSymbol.

In order to apply a Construction it must be self-contained. Before we define what self-containment means in this context, we have to introduce two definitions:

Definition 7 (*Loose Symbol*)

If for a Construction $c \in O_c.C$, there is a Symbol $y \in c.Y$ such that

$$y \in symb(c.S_E \cup c.S_C) : \neg(y \in c.Y) \tag{3.12}$$

This means that if y is part of a Statement of a Construction c but is not part of the Symbols $c.Y$ of the Construction, the Symbol is called a *loose Symbol*. We introduce a new predicate

$$c \in O_c.C \tag{3.13a}$$

$$lSymb(x, c) \tag{3.13b}$$

which for a given Symbol x and a Construction c evaluates if x is a loose Symbol of c.

With this definition we can now define what a self-contained Construction is.

Definition 8 (*Self-contained Construction*)

A Construction $c \in O_c.C$ is called self-contained, if

$$selfContained(c) \Leftrightarrow \forall y \in c.Y : \neg lSymb(y, c) \quad (3.14)$$

This means that a Construction is called self-contained, if it does not contain any loose symbols.

Self-containment is required for Constructions. It specifies that a Construction $c \in O_c$ can not reference any information outside its own scope, i.e., only Symbols which belong to $c.Y$ can be referenced by the Statements within $c.S_C \cup c.S_E$. If c would, however, contain a loose Symbol, the evaluation would become more complex and indeterministic. The reason is that it would be impossible to predict which value the loose Symbol should be instantiated with (this is explained later in section 4.2). Hence, only self-contained Constructions are considered for the evaluation process.
The last scope, which is of relevance to the thesis, is the InterpretationScope. We specify it as follows:

Definition 9 (*InterpretationScope*)

Let

$$O_i \quad (3.15)$$

be the InterpretationScope, which contains all InterpretationModels.

The specification of an InterpretationModel can be seen in definition 10.

Definition 10 (*InterpretationModel*)

Let

$$m := \{m.E, m.R\} \quad (3.16)$$

be an InterpretationModel, where $m.E$ contains all InterpretationElements and $m.R$ is the set of all InterpretationRelations within m. $m.E$ consists of the subsets

$$m.E := \{m.E_{sem}, m.E_{syn}, m.E_{con}\} \quad (3.17)$$

where $m.E_{sem}$ contains all SemanticElementInterpretations, $m.E_{syn}$ contains all SyntacticElementInterpretations and $m.E_{con}$ contains all ConstructionInterpretation. Fur-

ther let

$$m.R := \{m.R_{ass}, m.R_{int}\} \tag{3.18}$$

where $m.R_{ass}$ is the set of all AssociationInterpretations and $m.R_{int}$ contains the InterpretationRelations without AssociationInterpretations ($m.R_{int} := m.R \setminus m.R_{ass}$).

The specification of an InterpretationRelation is described in definition 11.

Definition 11 (*InterpretationRelation*)

Let $m \in O_i$ be an InterpretationModel. A relation $r \in m.R$ is defined as the set

$$r := \{r.src, r.trg\} \tag{3.19}$$

where $r.src, r.trg \in m.E$ and $r.src$ represents the source InterpretationElement of the InterpretationRelation r and $r.trg$ represents the target InterpretationElement of r. If $r \in m.R_{ass}$, it contains an additional attribute:

$$r := \{r.src, r.trg, r.typ\} \tag{3.20}$$

where $r.typ \in m.E_{con}$ and specifies the type of the AssociationInterpretation.

The last definition 12 specifies the different InterpretationElements.

Definition 12 (*InterpretationElement*)

Let $m \in O_i$ be an arbitrary InterpretationModel. Each SemanticElementInterpretation $e \in m.E_{sem}$ is defined as

$$e := \{e.sem\} \tag{3.21}$$

where $e.sem \in O_{se}$ is a reference to the SemanticElement which the SemanticElementInterpretation represents. Each SyntacticElementInterpretation $e \in m.E_{syn}$ is defined as

$$e := \{e.syn\} \tag{3.22}$$

where $e.syn \in O_{sy}$ is a reference to the SyntacticElement which the SyntacticElementInterpretation represents. Each ConstructionInterpretation $e \in m.E_{con}$ is defined as

$$e := \{e.sem, e.con, e.E_{syn}, e.R_{ass}\} \tag{3.23}$$

where $e.sem$ is a single InterpretationRelation $r \in m.R_{int}$, whose target element $r.trg \in m.E_{sem}$. (note that if the corresponding SemanticElement within O_{se} should be referred to which $e.sem$ references, the expression $e.sem.sem$ will be used). Next, $e.con$ references a Construction $c \in O_c.C$ which the ContructionInterpretation represents. $e.E_{syn}$ is a subset of InterpretationRelations in $m.R_{int}$ which reference SyntacticElementInterpretations within $m.E_{syn}$. The attribute $e.R_{ass}$ is a subset of $m.R_{ass}$ and contains all those AssociationInterpretations $r \in m.R_{ass}$ which have the Construction-Interpretation e as its source, i.e., $r.src = e$.

These definitions specify how an InterpretationModel $m \in O_i$ can store the links between the natural language text (represented by SyntacticElementInterpretations in $m.E_{syn}$) and the ontology O on the other side.

3.5. Creating Knowledge

For every knowledge intensive system like SE-DSNL it is necessary to create the knowledge itself. This section describes how knowledge in case of SE-DSNL can be created and to which guidelines it must conform. For our evaluation we developed a modeling tool which automatically enforces these guidelines.

To create the semantic knowledge of an ontology there exist several different methodologies which describe how the information can be modeled manually, e.g., the "Ontology Development 101" guide [111]. We therefore do not describe the process in detail. Instead, this section gives a short overview of how information can be gathered from existing sources and how the information can be enriched with linguistic knowledge to create a valid SE-DSNL model.

The ideal case is that an OWL ontology exists within the current domain (if such an ontology is not available the expert can still model it manually). An OWL ontology can automatically be transformed into the SemanticScope of a SE-DSNL model. The process is specified in section 3.5.1. Next we describe in section 3.5.2 how the different elements within an SE-DSNL model can be represented using Forms, FormRoots and Constructions. Following, mapping complex syntactic structures to corresponding semantic information is explained in section 3.5.3, before in section 3.5.4 guidelines are presented to which a valid SE-DSNL model has to conform. It should be noted that several of the steps require a lot of elements at once, which means an enormous effort to the user. However, the process can be greatly facilitated with corresponding tool support.

3.5.1. OWL to SemanticScope Transformation

As mentioned before, the ideal starting point for the SE-DSNL approach is an existing OWL ontology. In this section we describe how the information is transformed into the SemanticScope.
OWL provides a multitude of different possibilities to create and reason with semantic information. In contrast, SE-DSNL tries to tackle the combination of natural language on one side and semantic knowledge on the other side. Therefore, it is not the goal of the transformation to preserve all the reasoning semantics of OWL in the transformation process. Its goal is that the classes of OWL, their properties as well as the corresponding individuals are transformed such that they can be expressed with linguistic information.

We previously mentioned that SE-DSNL does not yet differentiate between individuals and classes, therefore every element in the SemanticScope is a SemanticElement. This

means that there is a unique SemanticElement for every class, property and individual. In the following, we describe in detail how the transformation from OWL to SE-DSNL works by introducing a set of different methods which are required to describe the transformations.

The method

$addForm(x, a)$

specifies that a new Form based on the URI of an OWL class x is created and associated to the SemanticElement a. URIs normally are of the form 'Namespace:Name', however, only the *Name* part of the URI is used because it often represents a natural language word. The method involves the creation of a new Construction, a SemanticSymbol and a SyntacticSymbol. The SemanticSymbol references a, the SyntacticSymbol references the previously created Form and the Construction references both Symbols. A special variant of $addForm(x, a)$ is

$addLiteralForm(x, a)$

which instead of using the URI uses the concrete value of the literal x for adding a Form to a. The process also involves the creation of different Constructions like *addForm*. The method

$SemanticElement(x, a)$

creates a new SemanticElement (if it does not exist yet). For this the URI of an OWL class x is required. The reference to the either newly created or perhaps already existing SemanticElement is stored in the variable a. Further, the method calls $addForm(x, a)$ if a has been newly created.

$Association(x, y, z)$

adds a new association of type x from a ReferencableElement y to a ReferencableElement z. Another relationship type is created by the method

$Generalization(x, y)$

which adds a new Generalization between x and y such that $x \subseteq y$. The method

$$InstanceOf(x, y)$$

means that one SemanticElement x is an instance of y. The result of the method is a call to $Generalization(x, y)$, because SE-DSNL does not yet support a clear differentiation between instance and concept level. In order to represent equality, the method

$$EquivalentTo(x, y)$$

creates the elements which are required to represent an equality between the elements x and y. Its type points to a SemanticElement s representing the equality concept. The concept is further mapped to different linguistic expressions for equality. Finally, the methods $Association(s, x, y)$ and $Association(s, y, x)$ are executed, which insert an Association of type s between x and y and back. The last method is

$$InverseOf(x, y)$$

which indicates that a concept x is the inverse of y. Similar to the *EquivalentTo* method a new SemanticElement s representing the inverse concept is created and mapped to corresponding linguistic expressions. Finally, the methods $Association(s, x, y)$ and $Association(s, y, x)$ are executed, which insert an Association of type s between x and y and back.

We use these methods to formally specify the transformations to SE-DSNL. OWL provides a huge set of different elements and features which can be used to model ontologies. However, many experts only require a small subset of the OWL features to build an ontology. We therefore decided, to only transform the mostly used elements to SE-DSNL in a first step. To select these elements we analyzed different publicly available ontologies [11] [112] and selected only such elements which were used in those ontologies. This lead to the selection of elements which can be seen in table 3.1. We explain these different transformations in the following.

1. *Class* is the central element of an OWL TBox. If a class x is found in the transformation process (as described in the table 3.1), a SemanticElement, based on the URI of the class will be created. The reference on the SE-DSNL element representing x it stored in the variable a. Further, a form based on the URI of x represents a.
2. A *Datatype* in OWL is an element which references a set of data values. The element is represented with the same mechanisms as an OWL class.

3. OWL defines properties between classes by using the range and domain of the corresponding ObjectProperty, which are called *ObjectPropertyRange* and *ObjectPropertyDomain*. In order to transform the information both domain and range are needed at the same time. This type of information is represented within SE-DSNL in the form of an Association between two SemanticElements. In case that there are multiple entries in either the domain or range specification (in the table 3.1 represented by z and y in row three), there is an Association from every SemanticElement, representing the items in y, to every SemanticElement, which represents the items in z.

4. The *ObjectPropertyAssertion* associates pairs of individuals within OWL. The element can be translated to SE-DSNL directly by associating the corresponding SemanticElements, which represent the source and target individuals.

5. The *DataPropertyDomain* specifies in which class a DataProperty is used in. In order to specify the structure in SE-DSNL an Association of the SemanticElement 'Attribute' is added between the SemanticElement, which represents the class, and the SemanticElement which represents the data element.

6. The *DataPropertyRange* defines the type a specific DataProperty represents. The information is transformed to the SE-DSNL meta model by associating the SemanticElement of the DataProperty to a SemanticElement which represents the data type of the ObjectProperty.

7. The *DataPropertyAssertion* adds a literal z to an individual y by using a previously defined data element (x). To recreate the structure in the meta model the following is done: A SemanticElement c is created which receives the value of the literal z as its name. Next c is marked as an instance of the SemanticElement a, the representation of data element. Finally the SemanticElement b, which represents the domain class y of the data property, is being associated to c by the SemanticElement type d.

8. The *ClassAssertion* marks an individual as belonging to a specific class. Here the individual y is represented by the SemanticElement b. In order to specify that b is an instance of a (the SemanticElement representing the class x) the *InstanceOf* method relates both elements.

9. A *SubClassOf* axiom specifies that one class x is more specific than another class y. The same information is represented in the meta model by using a Generalization between the corresponding SemanticElements.

10. The *EquivalentClass* axiom specifies that x and y are semantically equivalent. This can be translated to SE-DSNL by retrieving the SemanticElements which represent them and relate them accordingly, using the *EquivalentTo* method.

11. The OWL element *SubDataPropertyOf* specifies that for two DataProperties x, y $x \subseteq y$, i.e., the DataProperty x is more specific than or equal to y. In SE-DSNL, everything is expressed using SemanticElements. This means that in order to transform the information to this meta model, a Generalization is inserted between the corresponding SemanticElements.
12. *EquivalentDataProperty* specifies that two DataProperties x and y share the same semantic meaning. Similar to the EquivalentClass axiom, the corresponding SemanticElements a and b are related by using the *EquivalentTo* method, which inserts two Associations, representing equality, between a and b.
13. The *SubObjectPropertyOf* element specifies that one ObjectProperty x is more specific than or equal to another one y, i.e., $x \subseteq y$. The transformation to SE-DSNL is the same as for the SubClassOf element.
14. *EquivalentObjectProperties* defines that two ObjectProperties x and y are the same. The transformation to SE-DSNL is the same as for EquivalentDataProperty.
15. The element *InverseObjectProperties* specifies that one ObjectProperty x represents the inverse of another ObjectProperty y, i.e., $x \equiv \neg y$. The structure can be transformed by executing the method *InverseOf* with the corresponding SemanticElements a and b. This leads to two new Associations of a type which represents inversion, between a and b.

OWL provides more elements for which no transformation to SE-DSNL has been provided yet. If another element needs to be transformed, this can in general be done as follows:

1. For every information in the structure that one wants to transform, a SemanticElement has to be created.
2. A SemanticElement t needs to be retrieved which represents the type of the relation (e.g., as in the DataProperty transformation in table 3.1 an 'Attribute' SemanticElement is used to represent the type). In some cases, however, (e.g., in the 'SubClassOf' transformation) a specific relation is required which also exists within SE-DSNL (here the Generalization). In this case, no SemanticElement needs to be created.
3. The Association of type t has to be inserted between the SemanticElements, as long as no specific relation exists. If a specific relation exists, insert this one between the SemanticElements.
4. If some of the information in OWL contains a natural language label, which should also exist in the SE-DSNL model, a corresponding Form has to be added to the

Table 3.1.: OWL to SE-DSNL Transformation

ID	OWL Element	Transformation
1	Class(x)	SemanticElement(x, a) addForm(x, a)
2	Datatype(x)	SemanticElement(x, a) addForm(x, a)
3	ObjectPropertyRange(x, z) ObjectPropertyDomain(x, y)	SemanticElement(x, a) SemanticElement(y, b) SemanticElement(z, c) Association(a, b, c)
4	ObjectPropertyAssertion(x, y, z)	SemanticElement(x, a) SemanticElement(y, b) SemanticElement(z, c) Association(a, b, c)
5	DataPropertyDomain(x,y)	SemanticElement(x, a) SemanticElement(y, b) SemanticElement('Attribute', c) Association(c, b, a)
6	DataPropertyRange(x,y)	SemanticElement(x, a) SemanticElement(y, b) SemanticElement('Type', c) Association(c, b, a)
7	DataPropertyAssertion(x,y,z)	SemanticElement(x, a) SemanticElement(y, b) SemanticElement(z, c) addLiteralForm(z, c) InstanceOf(c, a) SemanticElement('Attribute', d) Association(d, b, c)
8	ClassAssertion(x,y)	SemanticElement(x, a) SemanticElement(y, b) InstanceOf(b,a)
9	SubClassOf(x,y)	SemanticElement(x, a) SemanticElement(y, b) Generalization(a, b)
10	EquivalentClasses (x,y)	SemanticElement(x, a) SemanticElement(y, b) EquivalentTo(a, b)
11	SubDataPropertyOf (x,y)	SemanticElement(x, a) SemanticElement(y, b) Generalization(a, b)
12	EquivalentDataProperty (x,y)	SemanticElement(x, a) SemanticElement(y, b) EquivalentTo(a, b)
13	SubObjectPropertyOf (x,y)	SemanticElement(x, a) SemanticElement(y, b) Generalization(a, b)
14	EquivalentObjectProperties (x,y)	SemanticElement(x, a) SemanticElement(y, b) EquivalentTo(a, b)
15	InverseObjectProperties (x,y)	SemanticElement(x, a) SemanticElement(y, b) InverseOf(a, b)

specific SemanticElement.

It should be mentioned again that SE-DSNL tries to capture the mapping of ontological to linguistic knowledge only. Certain axioms, which mainly represent information required for reasoning processes, therefore do not have to be transformed.

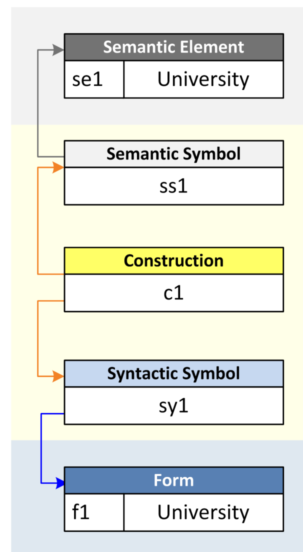


Figure 3.10.: 'University' Construction Example

3.5.2. Linguistic Representation of SemanticElements

One of the most common tasks is the linguistic representation of single ontological elements. To do this within SE-DSNL, the user can not simply add a Form to a SemanticElement, but has to create an additional Construction as well as some Symbols. An example can be seen in figure 3.10. Here, a new Construction *c1* was created which references two Symbols *sy1* and *ss1*. The first Symbol is a SyntacticSymbol and actually references the Form which represents the SemanticElement 'University'. The other Symbol *ss1* is a SemanticSymbol and references the aforementioned SemanticElement. This type of Construction (i.e., one which simply maps one SemanticElement to one Form) is called *Atomic Construction* or *Mapping Construction*. Such Constructions always only reference two Symbols, one syntactic and one semantic Symbol.

Linguistically representing one SemanticElement already seems like a lot of effort, because five elements are required in total (one Construction, one SemanticElement, one SyntacticElement and two Symbols). However, the process can be greatly facilitated with corresponding tool support, i.e., CodeCompletion or Intellisense. In the following we mostly represent Constructions in textual forms like the one which can be seen in table 3.2. The table represents the Construction in figure 3.10. The first row "Symbols" contains all the Symbols which the Construction references. The syntax of a Symbol first represents the specific Symbol type (e.g., SemanticSymbol or SyntacticSymbol). Next, the element it references is mentioned (in the example this is "University"; It should be mentioned that both Symbols reference different elements although the name is the same). The final term represents the name with which the corresponding element is used within

the Construction (here *sy1* and *ss1*).

Table 3.2.: Textual Representation of Construction *c1* 'University'

Name	Content
Symbols	SemanticSymbol University ss1 SyntacticSymbol University sy1
ConditionStatements	
EffectStatements	

The purpose of the Construction is to simply map one SemanticElement to one Form. There are, however, more difficult expressions, e.g., "University of Augsburg". The example contains three words two of which have a definite concept associated with them ('University' and 'Augsburg'). If those two, however, are mentioned in a specific order with the little word "of" between them, the complete noun phrase represents a different concept, i.e., the one of the specific university which is located within Augsburg. To map the phrase to the corresponding SemanticElement more information is needed. The required Construction can be seen in figure 3.11, its textual representation is specified in table 3.3. We explain all the different steps that are required to build the structure in the following.

The Construction *c4* is at the center and represents the mapping between the noun phrase and the corresonding SemanticElement. It references three other Constructions which map the words 'University' (*c1*), 'of' (*c2*) and 'Augsburg' (*c3*). Next, the user specified a ConditionStatement *inOrder* which enforces the correct order of the words. The expert also defined an EffectStatement which inserts the correct SemanticElement (*ss4*) if the previously mentioned Constructions were found in the correct order. It is, of course, possible to add additional ConditionStatements or EffectStatements, depending on what the domain requires. Constructions which contain ConditionStatements or EffectStatements, are called *non-atomic Constructions* or *complex Constructions*.

Non-atomic Constructions can also be represented in textual and more simplistic form, which can be seen in table 3.3. In contrast to the textually represented Construction in table 3.2, the table also contains textually specified Statements. The textual representation of Constructions much more resembles the work which actually has to be done to create a Construction. There is still a lot of work involved, however, the SE-DSNL frameworks contains an Integrated Developer Environment (IDE) which supports the user in the process.

The development of a SE-DSNL model is a time consuming process during which the knowledge grows continuously. It might therefore happen that the user tries to enter a name for a SemanticElement or a Form which already exists. It is up to the expert to

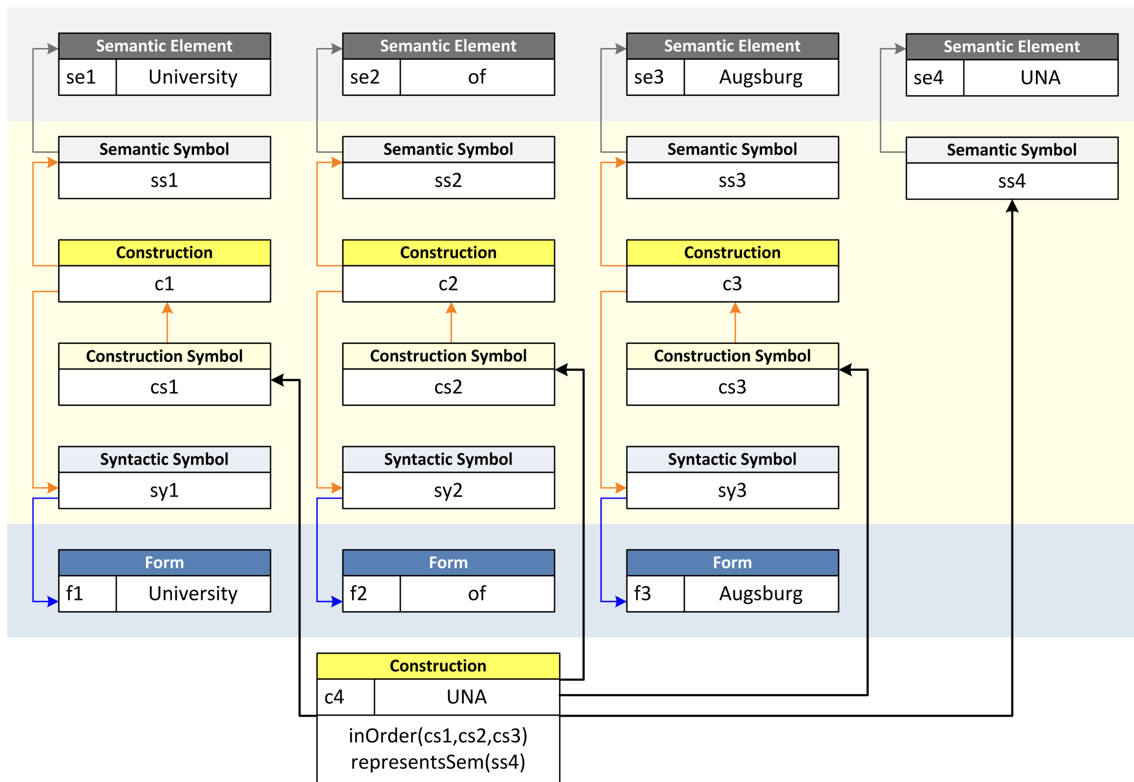


Figure 3.11.: 'University of Augsburg' Construction Example

Table 3.3.: Textual Representation of Construction *c4* 'UNA'

Name	Content
Symbols	ConstructionSymbol c1 cs1 ConstructionSymbol c2 cs2 ConstructionSymbol c3 cs3 SemanticSymbol UNA ss4
ConditionStatements	inOrder(cs1, cs2, cs3)
EffectStatements	representsSem(ss4)

decide if he / she is just trying to add a SemanticElement which already exists (which should be avoided) or the user is just adding an additional Form for a SemanticElement. It depends on the situation and the experience of the knowledge designer how the situation should be handled.

3.5.3. Mapping Syntactic and Semantic Structures

Previously we described how single SemanticElements can be represented linguistically. However, this is just one step towards fully mapping semantic and linguistic informa-

tion. SE-DSNL was especially developed to map syntactic to semantic structures. In the following, we describe how corresponding Constructions can be developed. The step is actually very similar to creating non-atomic Constructions. The main difference is that even more information has to be checked, e.g., information about the word order as well as the semantic relatedness of information within the SemanticScope.

For the example we chose to map a simple subject → predicate → object structure to the SE-DSNL knowledge. The result can be seen in figure 3.12 and table 3.4. First of all, three different Constructions are needed (*c1*,*c2* and *c3*), all of them referenced by a corresponding ConstructionSymbol (*cs1*, *cs1* and *cs3*). Next SyntacticCategories for identifying nouns (SyntacticCategory *f1* 'N' which is referenced by the SyntacticSymbol *sy1*) and verbs (SyntacticCategory *f2* 'V', referenced by SyntacticSymbol *sy2*) are needed. Two more SemanticElements have been created which represent the semantic knowledge, i.e., an 'Object' (referenced by the SemanticSymbol *ss1*) as well as an 'Action' (SemanticSymbol *ss2*). The SemanticElements are required because the ConditionStatements of Construction *c4* should validate that the meaning of the words exists within the SemanticScope. This can be done by using ConditionStatements. Basically the Constructions *c1* and *c3* both have to be of POS type noun. The corresponding validation is done by executing the ConditionStatements *isOfType*(*cs1*, *sy1*) and *isOfType*(*cs3*, *sy1*). These ConditionStatements check if the referenced knowledge of the second argument is contained within the first argument. In the example, the elements which are later assigned to either *cs1* or *cs3*, should therefore contain the SyntacticCategory 'N', which *sy1* references.

Table 3.4.: Textual Representation Construction NP VP NP

Name	Content
Symbols	ConstructionSymbol c1 cs1 ConstructionSymbol c2 cs2 ConstructionSymbol c3 cs3 SemanticSymbol Object ss1 SemanticSymbol Action ss2 SyntacticSymbol N sy1 SyntacticSymbol V sy2
Statements	inOrder(cs1, cs2, cs3) checkTriple(cs1, cs2, cs3) isOfType(cs1, sy1) isOfType(cs2, sy2) isOfType(cs3, sy1) isOfType(cs1, ss1) isOfType(cs2, ss2) isOfType(cs3, ss1) createTriple(cs1, cs2, cs3)

Other information constraints are checked similarly. For example, both *cs1* and *cs3* should contain information of the SemanticElement 'Object', i.e., *isOfType*(*cs1*, *ss1*) and *isOfType*(*cs3*, *ss1*). Construction *c2*, however, should be of the POS type 'V' (as referenced by the SyntacticSymbol *sy2*) and also reference a SemanticElement of type 'Action'. Next

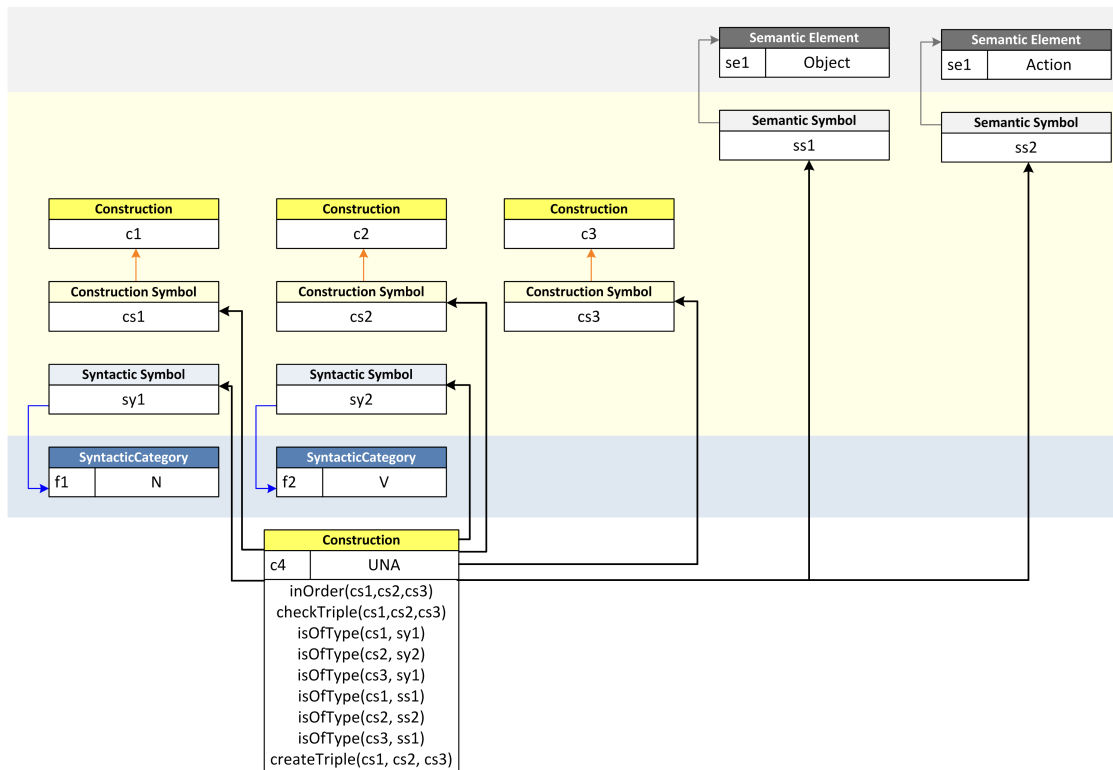


Figure 3.12.: NP → VP → NP Construction Example

the Constructions *c1*, *c2* and *c3* must appear in the order, i.e., $inOrder(cs1, cs2, cs3)$. Finally the ConditionStatement $checkTriple(cs1, cs2, cs3)$ checks if, based on the given semantics of the Constructions behind $cs1$, $cs2$ and $cs3$, knowledge is available within the SemanticScope.

If all ConditionStatements can be evaluated successfully, a triple is created, consisting of the semantic information which the instances of *c1*, *c2* and *c3* transport (a deeper introduction in the mechanisms and algorithms of the different Statements as well as the application of the Constructions in general is given in section 4.2 and following).

3.5.4. Modeling Guidelines

The development of a SE-DSNL model requires a lot of time to create all the knowledge. During the process there are many pitfalls which may lead to errors in the model. To circumvent some of these problems we developed several guidelines which can automatically be checked using a model validation, e.g., the data-flow based model analysis by Saad [113]. In the following, we describe the different guidelines which have to be considered in order to create a a technically sound SE-DSNL model. Hence, a definition

of different predicates is required (as seen in definition 13).

Definition 13 (*Guideline Predicates*)

Let

$m, n \in O_{se}$
$x, y, z \in O_{se}.E$
$a \in O_{se}.R$

be elements of the SemanticScope.
The first predicate validates if a Generalization between two SemanticElements exists:

$$gen(x, y) \Leftrightarrow x \subseteq y \quad (3.24)$$

This means that for two elements x and y $gen(x, y)$ specifies that x is equal to or more specific than y. It must be noted that the Generalization relationship in SE-DSNL is transitive, i.e.,

$$gen(z, y) \wedge gen(y, x) \Rightarrow gen(z, x) \quad (3.25)$$

Therefore, if z is a specialization of y and y is a specialization of x then z is also a specialization of x.
The predicate

$$equal(m, n) \Leftrightarrow a = b \quad (3.26)$$

validates if two elements m and n are equal.
The predicate

$$hasType(a) \Leftrightarrow a.typ \neq \varnothing \quad (3.27)$$

checks if the Association a has a type, i.e., an element from $O_{se}.E$. A variation of the method is defined in the following.
The predicate

$$isType(a, x) \Leftrightarrow a.typ = x \quad (3.28)$$

validates if x is the type of the association a.
The predicate

$$isSource(a, x) \Leftrightarrow a.src = x \quad (3.29)$$

checks if an element x is the source of Association a.
Finally

$$isTarget(a, x) \Leftrightarrow a.trg = x \tag{3.30}$$

validates if x is the target of Association a.

With these predicates we created a set of guidelines which are used to validate an ontology O:

1. $\exists e \in O_{se}.E \; \forall m \in O_{se}.E \; gen(m, e)$, i.e., all semantic elements must be part of a single hierarchy (note that *gen* is transitive as specified in definition 13). This means that there must be a path from every SemanticElement within the SemanticScope to a single root element by using Generalizations only. The root SemanticElement may not have any further outgoing Generalization links. This is due to some of the algorithms which measure and identify relations between all available concept.

2. SemanticElements are allowed to use multiple inheritance, i.e., one element can inherit from multiple other elements (e.g., the element 'Mr. Schmid' can be both a 'Man' and a 'Driver').

3. $\forall e, f \in O_{se}.E \; \neg(gen(e, f) \wedge gen(f, e)) \vee equal(e, f)$: Generalization relations are not allowed to form circles, i.e., there must not be a single element f within the SemanticScope from which one can return to the element if it uses the outgoing Generalization relations from the element e.

4. $\forall a \in O_{se}.R \; hasType(a)$: Each Association must have a type. As the type of a relation between two elements is one of the most important features which differentiates simple graphs from ontologies this is required for the approach.

5. The type of an Association must be part of a different Generalization branch than the source and target elements of the Association:

   $\forall a \in O_{se}.R \; \exists e_s, e_t, e_y \in O_{se}.E \; isSource(a, e_s) \wedge isTarget(a, e_t) \wedge isType(a, e_y) \wedge \neg(gen(e_s, e_y) \vee gen(e_y, e_s) \vee gen(e_t, e_y) \vee gen(e_y, e_t))$

   The constraint defines that the source and target SemanticElements of an Association are not allowed to be either a child or a parent of the SemanticElement which represents the type of the Association. The restriction is necessary for the semantic spreading activation (see section 5), which otherwise would have problems with spreading the tokens correctly and identifying the best matching Associations.

6. It must be noted that no matter what type of syntactic representation is used in a natural language sentence (i.e., either active or passive verb forms) the information

within the SemanticScope must always be modeled either in an active or passive representation (here everything is modeled in an active representation). Only this way a consistent mapping between the syntactic structure on one side and a semantic structure on the other side can be guaranteed.

The SyntacticScope is all about creating the information which is necessary to represent both the lexical and syntactic information. There are also different guidelines which the Scope must adhere to.

1. Every Form or FormRoot may only contain one single word, i.e., strings like 'University of Augsburg' are not valid values for either a Form or FormRoot. Instead three different Forms 'University', 'of' and 'Augsburg' must be created. Representing compound words is the task of the ConstructionScope and has been shown in section 3.5.2

2. The SyntacticCategory elements build a bridge to the parsing results of the syntactic parsers. Therefore, each SyntacticCagetory should represent either a POS tag or a syntactic category which is returned by the syntactic parser. For example, if the parser outputs 'VMFIN' there should be a SyntacticCategory in the SyntacticScope with the value 'VMFIN'.

3. In contrast to SemanticElements, SyntacticCategory elements do not have to be part of the same Generalization hierarchy.

4. It can be helpful to cluster the elements, depending on how fine grained the modeling of the Constructions should be. Hence, as an example, all POS tags representing the different kinds of verbs (e.g., modal and auxiliary verbs) can be clustered under one more general element.

Similar to the SemanticScope and SyntacticScope, the elements within the ConstructionScope should follow specific constraints.

1. Constructions have to be part of exactly one Generalization hierarchy with exactly one root element (the same as in guideline 1). Multiple inheritance is not allowed.

2. To map a single word to a specific element a Construction *c1* has to reference a SyntacticSymbol (*sy1* in figure 3.10), which references the Form *f1*. Further, the SemanticElement *se1* is used by referencing the SemanticSymbol *ss1*. Atomic Constructions therefore always only reference two Symbols, i.e., one SyntacticSymbol and one SemanticSymbol.

3. The development of a consistent ConstructionScope (i.e., a ConstructionScope which covers all the linguistic requirements of a specific domain) can be difficult and time consuming. Therefore, the user must be clear about the required gran-

ularity and detail. If he / she wants a very fine grained detail he must of course model his Constructions as detailed as possible (therefore the example from figure 3.12 would most probably be insufficient). However, this leads to a decreased recall which in turn means that more Constructions must be created for every possible syntactic structure. On a more abstract level, however, one can focus on modeling Constructions which have a high recall but might lack a certain degree of precision. Finding a right trade off between both sides is difficult and mostly depends on the task at hand.

4. A problem which is difficult to track is modeling Constructions in such a way that the analysis process wont be caught in an infinite loop. Figure 3.13 shows the most simple Construction structure which could lead to an infinite loop during the analysis process. The Construction simply references a ConstructionSymbol *cs1* which again references the same Construction (if modeled as a production rule it would look like $A \rightarrow A$. This type of Construction without any further restrictions could be repeated endlessly. It is, therefore, necessary to model the ConstructionScope in a way that infinite loops won't happen. However, this is a difficult task, especially in more complicated situations, as an infinite loop could also be the result of an EffectStatement. Hence, experts have to be very careful during the process.

5. Constructions have to be self-contained as specified in definition 8, therefore

$$\forall c \in O_c \; selfContained(c) \tag{3.31}$$

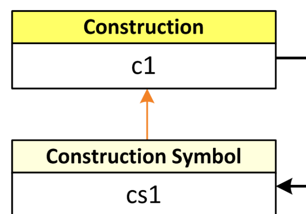


Figure 3.13.: Construction leading to an infinite loop

In order for the algorithms and mechanisms of the thesis to function properly, a valid SE-DSNL Ontology is required.

Definition 14 (*Valid Ontology*)

An ontology O is called valid if it complies to all guidelines of section 3.5.4.

3.5.5. Multilingual Representation

In section 3.2 several differences between languages were mentioned. We describe in this section that SE-DSNL is capable of representing any type of language. Basically, there are two types of information that are relevant for mapping the lexical and syntactic information of a language to semantic knowledge:

1. The first type is the order of the words. This is obviously true for English, French, German etc. as the order of the words, e.g., determines the roles of the nouns. Although the order of words is not as important in languages like Hebrew or Japanese, they still require a preposition or postposition to mark nouns as being the subject or object of a verb. A *pre*position comes in front of a word and a *post*position comes after a word, therefore also requiring a local order.
2. The second way of marking the roles of words within a sentence is by changing their morphology, i.e., altering either the endings (Latin, Russian) or even the complete word (as it is the case in Hebrew, where verbs are constructed of three consonants, which are put into a 'template' [3], e.g., the Arabic slm which means 'be at peace'; In order to express 'he was at peace' in the pattern 'XaXiXa' the 'X's are exchanged with the corresponding consonants, which leads to the word "salima").

Both information types are very important for classical statistical parsing ([114], [115], [116]). The same is true for mapping semantic knowledge to language, because lexical as well as syntactic information has to be identified correctly before it can be mapped to its corresponding semantic counterpart. In SE-DSNL both types of information are best identified by using Statements. First of all, the order of words can only be identified by using a corresponding Function (we describe the one we developed in section 4.3.1). Using a specific Function makes it easier to adapt to a specific syntax tree parser, because different parsers also yield different parsing trees. Regarding the morphology there are two different ways how this could be handled. The direct way would be to model each morphological form of a verb as a single Form and associate it to its corresponding SemanticElement. This, however, would be a very time consuming task, especially in languages with hundreds of different variations (such as Arabic [3] [117]). A more efficient way could be to directly incorporate external knowledge by using a Function. Next, the Function could parse a given word, identify its grammatical function and map it to a corresponding SemanticElement or semantic structure. This is more flexible and time efficient than the approach presented in, e.g., LexInfo [108] [12], where each morphological form would have to be modeled manually. Further, the Function-based approach could keep the SE-DSNL model smaller, as less information would be directly encoded within it.

3.5.6. Conclusion

The methodology presented in this section has been used for all case studies, examples and evaluations in the thesis. However, as the concept represents a generic approach it can also be used in a varying number of scenarios, both semantically as well as linguistically. The current focus lays on creating triples all of which represent different facts about the domain. The Constructions which map the syntactic to semantic structures are therefore obviously focused on using the structures in the way they were intended. Still, it is possible to use different modeling guidelines. Those could for example try to center not on modeling the facts directly within the ontology, but instead focus on the concepts which represent the actions and processes that are available within a domain. The focus could be compared to method-argument structures, i.e., Constructions would center around the verbs and their arguments (see section 2.1.2.2). Therefore, instead of creating the facts as direct triples, the user should center on the actions and relate them to the different arguments which are required by those actions and processes (e.g., instead of modeling 'Person' → 'Drive' → 'Car' the user could model it like 'Drive' → 'Actor' → 'Person' and 'Drive' → 'ActingOn' → 'Car'). Depending on the context, this way of modeling might provide advantages in different situations or contain more precise information for syntactic structures. Also, combinations of different approaches are possible. However, different approaches might require new Constructions as well as new Statements which is one of the more complex phases in creating a knowledge base for SE-DSNL.

3.6. Related Work

Pohorec et al. [118] stress the importance of needing knowledge resources for NLP. They argue that even simplistic lexical knowledge bases improve the accuracy of NLP parsers. The improvement can even be enhanced if using semantic knowledge. Specifically ontologies can bring advantages to this field as they provide features like modularity or the description of terms with other ontologies. Several concepts exist which focus on how to create lexical representations of ontologies. Some of these provide multilingual representation capabilities.

SKOS [95] [96] is the Simple Knowledge Organization System. Its main goal is to enhance the possibilities of labeling elements within RDFS and OWL. It therefore introduces additional label variations like *prefLabel*. The additional information can be marked with a language related tag, thereby enabling a simplistic multilingual representation of RDFS and OWL related knowledge. However, it lacks the capabilities to describe more complex linguistic information in detail. Especially, requirements 1 to 4 from section 3.2 can not be fulfilled. This means that SKOS is only suited in very narrow scenarios for natural language processing related tasks.

The Lexical Markup Framework [98] defines a meta model which is suited "for the construction of NLP lexicons" [97]. Its structure supports the creation of deeper linguistic structures, thereby fulfilling requirements 1,2,3 and 5. However, it fails on delivering a connection to deeper semantic meaning, e.g., from an ontology. It therefore fails in fulfilling requirement 4.

Montiel-Ponsoda et al. [119] [120] developed the LIR concept which allows ontologies to be expressed in different languages. It focuses on the lexical representation only and allows expressing the semantic information of one ontology in multiple languages. Therefore, it provides a meta model which enables a user to create a lexical database source, which can be linked to conceptual information. However, it is missing the capabilities of modeling complex linguistic patterns or representing morphological and syntactic decomposition. It can therefore not fulfill requirements 1 and 3.

A very expressive model has been developed already in the late 80ies by Bateman et al. The model was called the generalized upper model [121] [122]. Its design is built around introducing a linguistic knowledge organization layer which maps linguistic to semantic information. The abstraction layer is based on the Penman Upper model [123] and Merged Upper Model [124]. It provides many of the features of newer concepts in the domain, however, there is no information about how morphological variations can be captured.

LexInfo [108] [12] is one of the most expressive standards with regard to linguistically

representing ontological knowledge (a deeper introduction is available in section 2.2.3). It is built upon LexOnto [125], LingInfo [126] and further, in its latest version, also integrated the information from Lemon [127]. Also, it incorporates information of the Lexical Markup Framework [98] for a better mapping from syntactic to semantic structures. It's model has been realized as an OWL ontology[1]. It allows an exact and detailed representation of lexical information and its mapping to the corresponding ontological knowledge. Further, syntactic variations and features as well as subcategorization frames and a simplified structural mapping to an arbitrary ontology can be defined. The main differences between SE-DSNL and LexInfo are that SE-DSNL provides a more open and flexible way of modeling and mapping linguistic as well as structural mapping information. Also, morphological variations can be captured in SE-DSNL using Statements and Functions. Further, LexInfo has more restrictions on the way that it can specify structural mappings, i.e., the mapping of a verb to a specific ObjectProperty. Basically, if a verb requires more than two arguments, it can not be mapped to a single ObjectProperty, because the Domain and Range of a property are not enough. Instead a verb with three or more arguments must be mapped to more complex ontological structures. Such situations can be handled with the SE-DSNL approach. An advantage of LexInfo over SE-DSNL is its direct integration within OWL, which allows the definition of axioms as well as the direct verification of an ontology by using a reasoner.

[1] http://lexinfo.net/

4

Semantic Interpretation of Natural Language

4.1. Introduction

In the previous chapters, the problems, objectives and basics of the thesis have been formulated, i.e., how natural language text can be mapped to ontological knowledge. We further introduced the SE-DSNL meta model on which the rest of this thesis is based on. The chapter presents the algorithms which interpret natural language text based on a SE-DSNL model. The core of the approach is a generic concept of how Constructions and the information within them can be applied to a given text. It allows the incorporation of semantic knowledge during the application process itself due to a flexible and extensible structure. In order to specify their behavior we further introduce a set of Functions for either analyzing syntactic structures or querying information from the SemanticScope.

The advantages of our approach are as follows:

1. The algorithm for applying Constructions is concurrent (as defined by Ben-Ari [128]). The approach has the advantage that it can easily be optimized for multi-core systems (note that the prototypical implementation of SE-DSNL does not support multithreading). The way the application algorithm has been designed, also allows Constructions to access the results of other Constructions. This means that our approach does not employ a one-way-pipeline. Instead information which has been added at runtime by one Construction can lead to new information from other Constructions. The process finishes if either no more Constructions can be applied or one of the heuristics terminates the process. The approach ensures that an optimal result can be found.

2. The different Functions have been specified such that they are optimal for German and English. New Functions for other languages or other types of semantic knowledge can just as easily be developed and integrated into the overall process. SE-DSNL therefore provides an optimal adaptability to different circumstances.

3. Functions can introduce new information from the SemanticScope in the parsing result at runtime, i.e., immediately during the application of Constructions (many OBIR systems only use the semantic information for validating their results). This allows the SE-DSNL concept to optimize the initial input by integrating ontological knowledge into the final result. The approach can be compared to the process how humans 'enrich' vague or imprecise information with implicit knowledge.

The chapter is structured as follows: Section 4.2 describes how information from a SE-DSNL model are used to parse natural language text. The approach is based upon a syntax tree to which the Construction application process itself is aligned. In section 4.3, a set of different Functions is specified which can be used to define Constructions. We finally delimit SE-DSNL from other approaches in section 4.4.

4.2. Semantic Interpretation

One requirement for this thesis was a generic applicability to different languages, i.e., different lexical, syntactic as well as semantic structures. On a static level, the requirement has been fulfilled with the meta model which was presented in section 3. The mapping between the different information types can be created with Constructions. These capture the definition of how existing information can be combined, i.e., how existing linguistic as well as semantic information should be put together to form new knowledge. The problem with such a generic approach is that it can require a lot of computational power: If there is no easily identifiable structure available within the initial input natural language text, the structure must somehow be extracted. To facilitate the process we decided to introduce informations and structures which can be created for many different languages by currently available NLP components, i.e., POS tags and syntax trees. The information guides the overall process. The mechanism how syntax trees are used within SE-DSNL are described in the following sections. Next, the process of how Constructions are instantiated is specified.

4.2.1. Syntax Tree Alignment

Parsing text itself is a difficult task. There are multiple NLP components available to syntactically parse text, i.e., they assign POS tags to each word and from that try to infer a potential tree like structure which combines a set of POS tags (the leafs of the tree) by grouping them in syntactic categories. The tree structure can be very helpful as it marks the different constituents and phrases of a sentence and how they are related to each other.

Figure 4.1 contains two example syntax trees. On the left side the syntax tree for the first sentence is presented, whereas on the right side the syntax tree for the second sentence can be seen. The leafs of the tree consist of the POS tags whereas the inner nodes represent the different syntactic categories (the abbreviations are based on those in the Penn Treebank; An overview can be found in the appendix section A.1).

The parsing results contain a problem, as the 'I' within the first sentence was not capitalized, therefore the parser marked the word as being a foreign word (FW). Depending on the domain it might, however, be necessary to have a certain robustness towards text with spelling errors as well as not 100% correct grammatical structures. Another problem is that depending on the language, which SE-DSNL should be used with, different parsers might be required which deliver results of different quality. An example of how slight variations in a sentence can have huge effects on the parsing result can be seen in figure 4.2. In contrast to the first sentence from figure 4.1, the word 'just' was removed.

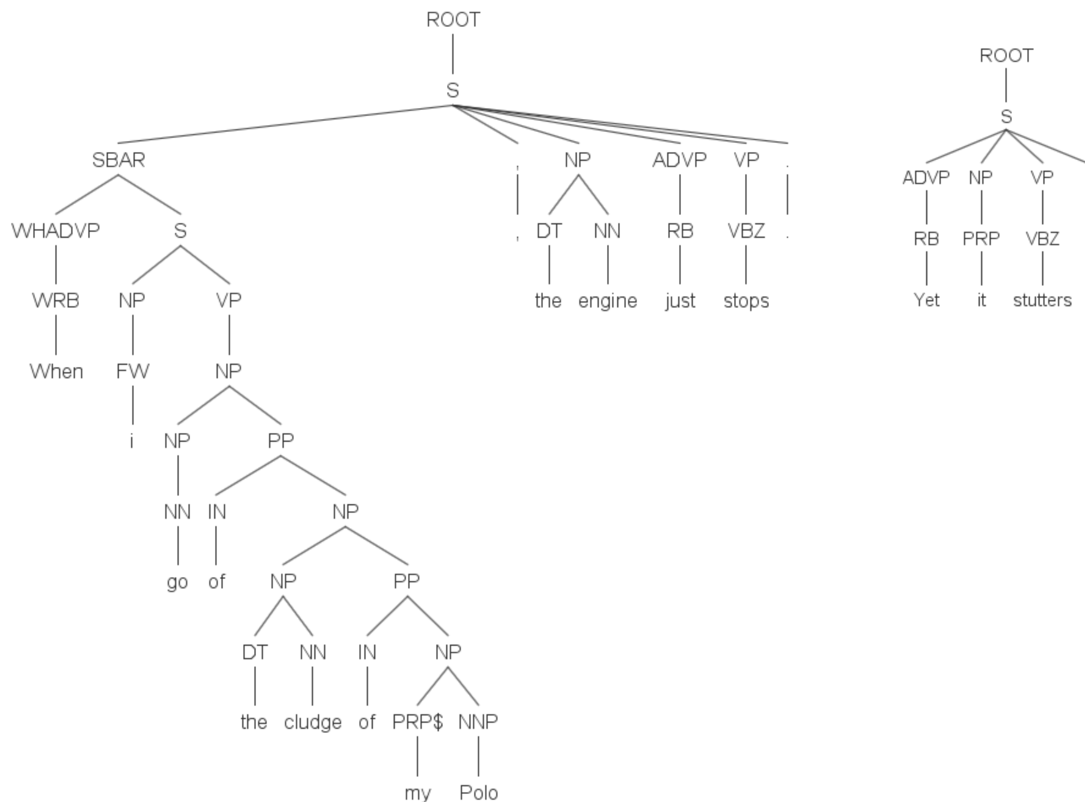


Figure 4.1.: Syntax trees for the two sentences "When I go of the cludge of my Polo, the engine just stops. Yet it stutters.", created with Stanford Parser V1.6.8 and English PCFG grammar

As a result the parser changed the complete structure of the syntax tree. In this new structure the words from 'i' to 'stops' form a subsentence. Additionally 'when' and the subsentence supposedly build a direct question as indicated by SBARQ.

It is obvious that the syntax tree as a whole is wrong. Yet major parts of it are correct, e.g., the noun phrases. This still allows us to use a syntax tree as a guidance for our process in order to enhance computational performance and overall precision. The approach is based on the idea that Constructions are built by experts. The main motivation of an expert introducing SE-DSNL in his domain is that he can identify all the information which is necessary for his / her task. Therefore, he / she only creates valid Constructions, i.e., Constructions which identify relevant structures. On the other side syntax parsers try to create valid syntactic parse trees from natural language text. Although some of the results may contain errors there are most probably several aspects of a tree which are still correct (as described in the previous example). We therefore make the following assumption:

Assumption 1 (*Syntax Tree Alignment Assumption*)

If a Construction can be applied perfectly to a node of the syntax tree we assume

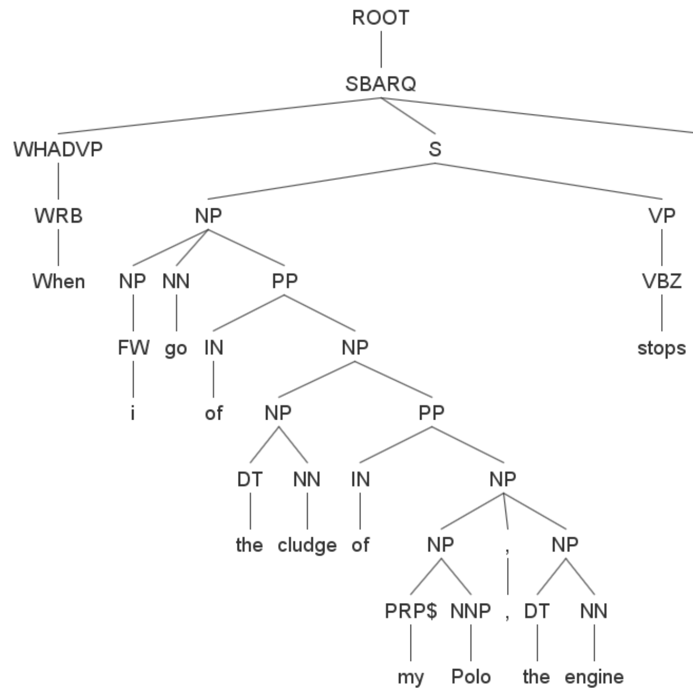


Figure 4.2.: Syntax tree of the modified first sentence ('just' has been removed), created with Stanford Parser V1.6.8 and English factored grammar

that this tree structure is itself an optimal solution for the specific natural language text. Hence, the result of the Construction is optimal. A Construction matches a node perfectly if the Construction can make use of all the information available within this node and its children. This means that a Construction is allowed to not only reference the information of a single node but also all of the information of its children. This way correct information gets rated higher than the remaining possibilities.

What the assumption means can be seen in the following example. Figure 4.3 shows a small syntax tree and three different Constructions which could match the structure: Two one-argument Constructions (one which needs the noun before the verb and the other one the verb before the noun) as well as one two-argument Construction (noun → verb → noun) all of which could be applied to the node 'S'. However, we would prefer the two argument Construction in this situation because it could make use of all the information available within the tree, i.e., it would reference both nouns (as it needs a subject and an object) as well as the verb. Therefore, as one Construction is available which matches the node perfectly, the Construction is preferred over the other ones. Without the syntax tree it would be more difficult to determine which Construction would be best suited in this specific situation because the constituent groups would not be available (it could be encoded within the Construction which, however, would require a lot of additional effort).

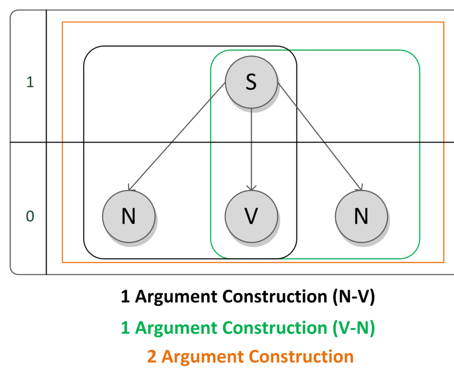


Figure 4.3.: Example of different possible Constructions matching a node in a syntax tree

Figure 4.4 shows a slightly different structure for the same text as in the previous example (it could be the result of a different parser). The POS nodes are the same, yet not all three of them are connected directly to the S node. Instead, there first is an 'NP' node at level one which connects the first two nodes 'N' and 'V'. All three are aggregated by the 'S' node at the top level. This time there is only one Construction which can be applied to the 'NP' node. The other one-argument Construction can only be applied to the 'S' node. However, there again is the better suited two-argument Construction which is capable of using all the information supplied within the syntax tree. Thus, despite the different syntactic structures, the Constructions can still be applied correctly to the syntax tree.

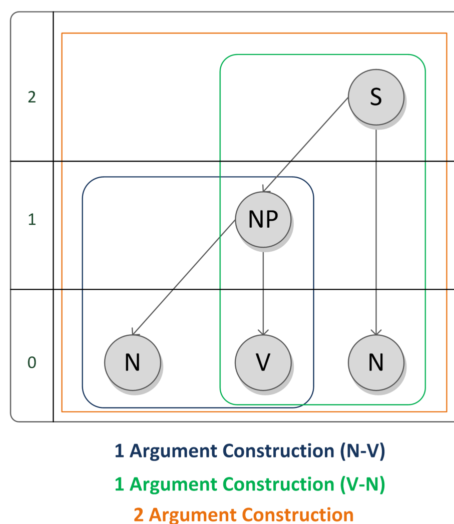


Figure 4.4.: Example of different possible Constructions matching a wrong syntax tree

As shown in the previous examples, the mechanism is based upon the assumption that the better a Construction fits a node in the tree the higher it is rated. The alignment process therefore starts at the leafs of the tree (the POS nodes) and works its way up the root node. For an example we take a look at figure 4.5. The deepest POS nodes here are N_3, V_2 and N_4. For each node the process tries to apply the available Constructions. However,

as there are no Constructions available for either of the single POS nodes, the process terminates and moves up one level to S_2. There, the two-argument Construction can be applied, which is a perfect and direct match. The result of the Construction application receives a high rating value.

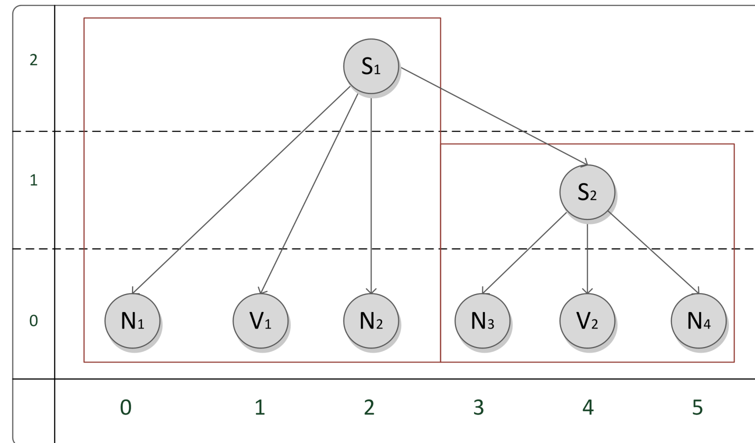


Figure 4.5.: Possibilities to match a two argument Construction to a syntax tree

Still this is not the root node, therefore the overall process can not terminate. Yet the nodes N_1, V_1 and N_2 have to be checked first (because they are below S_1). Again, there are no Constructions available therefore the process analyzes S_1. One condition of the two-argument Construction is that there should be as less space as possible between the noun → verb → noun. This means that only one of the following possibilities can receive the highest possible rating: $\{N_1, V_1, N_2\}, \{N_1, V_1, N3\}, \{N_1, V_1, N_4\}, \{N_1, V_2, N_4\}$ and $\{N_2, V_2, N_4\}$ ($\{N_3, V_2, N_4\}$ cannot be created as it already exists as part of the Construction of S_2). From these possibilities $\{N_1, V_1, N_2\}$ is picked because it best matches the conditions of the two-argument Construction (i.e., there is no space between the single words).

The remaining process steps consist of applying Constructions to a natural language text in accordance with its syntax tree structure. The process continues until the solution set is filled completely, i.e., no more Constructions can be applied. The internal structure then contains all relevant information for the final result. In a final step it is extracted which leads to a set of InterpretationModels.

The section presented the overall idea and assumption behind the syntax tree alignment. It explained that despite erroneous syntax tree parts the Construction application process can still yield correct results. It therefore provides a certain robustness towards failures within either the natural language input or not optimal syntax trees. In the following we explain how Constructions can be applied to a syntax tree node and how this leads to Construction instances.

4.2.2. Construction Application

The Construction Application process consists of applying Constructions to a given set of information. The ConditionStatements of a Construction are evaluated on the input set and, if successful, lead to a so called *ConstructionInstance*. In this element, each Symbol of the original Construction receives a concrete value. Further, the ConstructionInstance also contains the results of the EffectStatements which have been executed after the successful evaluation of the ConditionStatements. In the following, the process is specified in detail.

The process itself starts on a natural language input text.

Definition 15 (*Natural Language Text*)

Let natural language text be defined as a tuple

$$W := (w_1, .., w_n) \tag{4.1}$$

where W consists of a list of words $w_1..w_n$ in a specific order.

W is the input of the Construction application process. A word $w \in W$ can be mapped to one or more Forms $f \in O_{sy}.F$ using a string similarity algorithm. W is the input from which the syntax tree is created. The tree is defined as follows:

Definition 16 (*Syntax Tree*)

A syntax tree is defined as follows:

$$S := \{n_r, S_n, S_l\} \tag{4.2a}$$

$$S_n := (n_1, .., n_n) \tag{4.2b}$$

$$S_l \subseteq S_n \tag{4.2c}$$

where $n_r \in S_n$ is the root node of the tree, S_n is an ordered tuple of all nodes which are part of S and S_l is the set of leafs of S. Each node $n \in S_n$ may have only one parent node but can have an arbitrary number of children, which are connected to n with edges. Such nodes, which do not have any children, are contained within the leaf set S_l.

Note that if we talk about a node $n \in S_l$ in the following, n also contains the information of the word $w \in W$ that n has been mapped to.

In section 4.2.1 we described how the process is aligned to a syntax tree S. In order to realize the behavior it is best to parse the syntax tree bottom-up, i.e., the process starts at

the deepest node (which must be a member of S_l) of the syntax tree and works its way up. For each node the process tries to match all of its Constructions $c \in O_c$ to the node. For each matching Construction it creates one or more instances of this Construction. The process is described in algorithm 1. It receives the set O_c of all available Constructions as well as the syntax tree S as input parameters. It starts with creating a list S_O, which contains all the nodes of S in post order. S_O therefore contains all nodes in an order which ideally suits the intended bottom-up parsing process. Next the list is being iterated node by node. For each node s all available Constructions $c \in O_c$ are evaluated within *createInstances*(c, s). The method evaluates the ConditionStatements in $c.S_C$ and, if possible, instantiates the Construction. The newly created instances are added to the final solution set.

Algorithm 1 Mapping Phase Algorithm

Input: O_c: Set of all available Constructions
S: Syntax tree
Effect: Creates the initial Mapping Construction instances

```
1: procedure ALIGNCONSTRUCTIONTOTREE(O_c, S)
2:     S_O := createPostOrder(S)
3:     for all s in S_O do
4:         for all c in O_c do
5:             createInstances(c, s)
6:         end for
7:     end for
8: end procedure
```

The Construction application process is solely based on the available Constructions within O_c, which define what information should be identified and what results should be created from a specific input. We already mentioned that in order to store the information a Construction has to be instantiated. The corresponding data structure is called Construction Instance:

Definition 17 (*Construction Instance*)

Let c^I be a Construction instance which is defined as

$$c^I := \{c^I.c, c^I.Y^I, c^I.e, c^I.ref, c^I.rel, c^I.v, c^I.lvl, c^I.sen, c^I.sem, c^I.syn, c^I.s, c^I.int, c^I.id\} \quad (4.3)$$

The different elements of the set are specified as follows:

1. $c^I.c$ is the Construction which has been instantiated.

2. The set of symbols of $c.Y$ is instantiated and its concrete values stored in the set:

$$c^I.Y^I := \{y_1^I, .., y_n^I\} \tag{4.4}$$

i.e., for a Symbol $y_m \in c.Y$ its concrete value is stored in $c^I.y_m^I \in c^I.Y^I$.

3. $c^I.e$ contains the results of the EffectStatements $c.S_E$ (e.g., new instances or links between existing instances which have been created as part of the Construction instance).

4. $c^I.ref$ is a set which contains references to all Construction instances which are being referenced by the instance.

5. $c^I.rel \subseteq R^I$ (which is defined later in definition 4.9) stores typed relations from this to other Construction instances, i.e.,

$$\forall r^I \in c^I.rel\, r^I.c_s^I = c^I \tag{4.5}$$

where $r^I.c_s^I$ denotes the source of the relation r^I.

6. $c^I.v$ represents the likeliness of the Construction instance, i.e., a value between [0,1] which specifies how well the Statements of $c.S_C \cup c.S_E$ could be applied to the concrete values of the Symbol instances in $c^I.Y^I$.

7. $c^I.lvl$ is the level of the instance in correspondence with the syntax tree node where the instance was created (as seen in figure 4.7).

8. $c^I.sen$ contains a value which associates the Construction instance with the sentence it belongs to. The value is used for the identification of the Construction instance to avoid that multiple instances which have been applied to different sentences have the same identification.

9. $c^I.sem \in O_{se}$ specifies the SemanticElement which represents the semantic type of the Construction instance.

10. $c^I.syn \subseteq O_{sy}$ is a set containing all syntactic information about the node (e.g., a leaf instance can contain the word itself as well as its POS tag).

11. $c^I.s \in S_n$ is the node of the syntax tree S where the Construction instance has been created.

12. $c^I.int$ describes the interval which the Construction instance covers with regards to the text W that it has been applied to. This is indicated by the index of the leftmost and rightmost words $w_i, w_j \in W$ where $i <= j$.

13. A Construction instance c^I is further uniquely identified by its id:

$$c^I.id := (c^I.c, c^I.Y^I, c^I.e, c^I.sem, c^I.sen) \tag{4.6}$$

The Construction alignment process needs to store the different Construction instances, for which we defined two different sets (definition 18).

Definition 18 (*Construction Instance Sets*)

Let

$$C^I := \{c_1^I, .., c_n^I\} \tag{4.7}$$

be a set which stores all valid Construction instances, i.e., Construction instances which have been instantiated successfully after all ConditionStatements have yielded a positive result. Further, let

$$C_t^I := \{c_1^I, .., c_n^I, .., c_m^I\} \tag{4.8}$$

where $n \leq m$ and $C^I \subseteq C_t^I$. The set contains all Construction instances that have been created during the analysis of a natural language text, even those which were not applicable. The set is used for validating if a Construction has already been computed in a specific context or not.

In the Construction instance definition a set $c^I.rel$ was introduced which represents instantiated Relationships. Its exact specification is shown in definition 19.

Definition 19 (*Construction Instance Relation*)

Let

$$R^I := \{r_1^I, .., r_n^I\} \tag{4.9}$$

be the set which contains all instantiated Relationships, either Associations or Generalizations from $O_{se}.R \cup O_{se}.G$. Each relation $r^I \in R^I$ is defined as

$$r^I := \{r^I.c_s^I, r^I.c_t^I, r^I.c_y^I\} \tag{4.10}$$

where $r^I.c_s^I \in C^I$ is the source Construction instance of the relation, $r^I.c_t^I \in C^I$ is the target Construction instance of the relation and $r^I.c_y^I \in C^I$ specifies the concrete type of the instantiated relation.

Figure 4.6 shows how parts of the previous definitions are related. There, three different

Construction instances $i_1, i_2, i_3 \in C^I$ can be seen. The Construction instance relation $r^I \in R^I$ between i_2 and i_3 has been inserted as a result of Construction instance i_1. Hence, r^I is part of $i_1.e$ because it is the result of one of the EffectStatements in $i_1.c$. However, r^I is also part of $i_2.rel$. The reason is that it relates i_2 to i_3. Further both i_2 and i_3 are part of $i_1.ref$ because i_1 requires both to create the relation r^I.

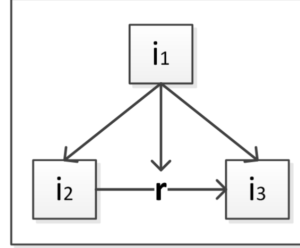


Figure 4.6.: Construction instances and Construction instance relation

The actual instantiation of a Construction is defined as follows:

Definition 20 (*Construction Instantiation*)

Let

$$children : S_n \to \mathcal{P}(S_n) \tag{4.11a}$$
$$instances : S_t \to \mathcal{P}(C^I) \tag{4.11b}$$

be two functions where *children* returns all the child nodes of a syntax tree node of S_n. The result of *instances* is a set of valid Construction instances, which have been created on one of the nodes in S_t (a Construction instance c^I belongs to a node $s_n \in S$ if $c^I.s = s_n$) Let

$$f : O_c, S \to \mathcal{P}(C^I_t) \tag{4.12}$$

be a function which, given a Construction from O_c and a single node from S, creates a set of Construction instances. Each Construction instance $c^I \in f(c,s)$ is derived from a Construction $c \in O_c$ and a syntax tree node $s \in S$ by first substituting the symbols in $c.Y$, i.e.,

$$\begin{aligned} &\phi : c.Y, X \to c^I.Y^I \\ &\quad X := s \cup children(s) \cup instances(children(s) \cup s \end{aligned} \tag{4.13}$$

where ϕ substitutes a symbol from $c.Y$ with an element from X. This leads to an instantiated Construction symbol in $c^I.Y^I$. In order to finish the instantiation, the instantiated symbols $c^I.Y^I$ have to be evaluated according to the Statements of the

Construction c:

$$eval : c.S, \mathcal{P}(c^I.Y^I) \rightarrow [0,1]$$
$$\forall c^I \in f(c,s) \forall s \in c.S_C \, eval(s, c^I.Y^I) \quad (4.14)$$

where $eval(s, c^I.Y^I)$ evaluates a set of instantiated Symbols according to the semantics of a specific ConditionStatement s and returns a value between 0 and 1. The results of the different Statement evaluations are combined using a fuzzy AND operator:

$$\wedge(x,y) := 1 - MIN(1; ((1-x)^p + (1-y)^p)^{\frac{1}{p}}) \quad (4.15)$$

where p is an arbitrary number. Depending on the outcome of the ConditionStatement evaluation, the Construction instance is added to C^I if the following holds:

$$\bigwedge_{s \in c.S_C} eval(s, c^I.Y^I) \geq T \quad (4.16)$$

where $\bigwedge$ computes the fuzzy-logic AND result of all *eval* values and T is a threshold. Therefore, if the evaluation of all ConditionStatements for an instance c^I is greater or equal than the threshold, c^I can potentially be added to C^I. If the evaluation value, however, is below T, the Construction instance will be added to C^I_t.
To finally decide, if an instance can be added to C^I, all EffectStatements $s \in c.S_E$ must be evaluated. This can lead to new information in $c.e$ as well as an element in $c^I.sem$. After the execution of all Statements $c.id$ is available which is used to check for a duplicate of c^I in C^I_t:

$$c^I \notin C^I_t \Rightarrow C^I := C^I \cup c^I \wedge C^I_t := C^I_t \cup c^I \quad (4.17)$$

If c^I is not part of C^I_t, it will finally be added to C^I as well as C^I_t.

The definition states that a Construction instance is basically a Construction whose Symbols have been substituted with concrete values. Those concrete values can be either references to syntactic information (in case of SyntacticSymbols) or other Construction instances (in case of ConstructionSymbols, which contain semantic information).

So far we explained what a Construction instance is. However, it was not yet described how a Construction instance is actually created. The algorithm can be seen in algorithm 2. It receives the Construction c (which should be instantiated) as well as the current syntax node s as arguments. Further, the constants X_1 and X_2 represent threshold values which are explained later. The algorithm starts by initializing the set of elements which is used to instantiate the single Symbols within a Construction instance. Hence, it first recursively collects all children of s (line 2) and stores them in the variable *syntaxNodes*.

Next, it collects all Construction instances of these syntax nodes and stores them in the variable *elems* (line 3). After that, a list containing the Cartesian product of *elems* is created (line 4), where $\#(conSymb(c.Y))$ returns the number of ConstructionSymbols within $c.Y$ (the Cartesian product is created by cloning the *elems* list as often as there are ConstructionSymbols; Next, all possible set combinations will be created which contain one element from each *elems* list).

Algorithm 2 Construction Instantiation

Input: c: A single Construction
s: Syntax tree node
X_1: Constant threshold value
X_2: Constant threshold value
Effect: A set of all Construction instances based on c and s

1: **procedure** CREATEINSTANCES(c, s, X_1, X_2)
2: $syntaxNodes$:= $\{s\}$ + getChildren(s)
3: $elems$:= instances($syntaxNodes$)
4: $cartProd$:= createCartesianProduct($elems$, $\#(conSym(c.Y))$)
5: $newInst$:= $False$
6: **for all** p in $cartProd$ **do**
7: c^I := new ConstructionInstance(c, p)
8: res := 1
9: **for all** s in $c.S_C$ **do**
10: res := $res \wedge$ s.evaluate(c^I)
11: **end for**
12: **if** $res \geq X_1$ **then**
13: **for all** s in $c.S_e$ **do**
14: s.evaluate(c^I)
15: **end for**
16: c^I.calculateLikeliness()
17: **if** $c^I \notin C_t^I$ && $c^I.v >= X_2$ **then**
18: $C^I := C^I + \{c^I\}$
19: $newInst$:= $True$
20: **end if**
21: **end if**
22: $C_t^I := C_t^I + \{c^I\}$
23: **end for**
24: **return** $newInst$
25: **end procedure**

The approach of checking every possible combination is necessary because it is not the instantiation algorithm which should introduce a bias towards the way a language is parsed, especially as the SE-DSNL concept should be applicable to as many different languages as possible. Therefore, only Constructions and their Statements are allowed to introduce restrictions on the way a language must be parsed. We will come back to this in section 7.3.3.

So far, the algorithm created the Cartesian product of all possible Construction instances. In line 5, the variable *newInst* is initially set to false. It will be set to true if new Construction instances have been created (line 19). Following, for every list of elements p in *cartProd* a new Construction instance c^I is being created (line 7). Then a temporary variable *res* is initialized which stores the value of the ConditionStatement evaluation (line 8 til 11). The results of the different ConditionStatements evaluations are added to *res* using a fuzzy-and operation (indicated by the $\wedge$ operator in line 10). After all ConditionStatements have been evaluated, the *res* value is checked against the threshold X_1 (line 12). If the value is greater or equal to the threshold, all EffectStatements will be executed (lines 13 to 15).

At this point, all information has been gathered which is relevant to the identification of the instance. Hence, the likeliness value $c^I.v$ is calculated in line 16. If the instance is new, i.e., it is not existing in C_t^I (line 17) and further its likeliness is greater or equal to the threshold X, it will be added to C^I (line 18). Further, the variable *newInst* is set to 1 because at least one new Construction instance has been created. Each Construction instance is also added to C_t^I (line 22), even if it is not a valid instance. Finally, the algorithm returns the value *newInst* which indicates if new instances have been created.

One method which has not been explained properly so far, is $c^I.calculateLikeliness()$ in line 16. The method sets the value of the attribute $c^I.v$, i.e., the likeliness of the instance. The value is used to rate different instances against each other and to validate if an instance is suited to be added to C^I. The computation of the likeliness can be seen in definition 21.

Definition 21 (*Construction Instance Likeliness*)

$c^I.v$ is defined as the likeliness of a Construction instance and specifies how well the instance is suited for the context in which it appears. The calculation of the value is based on several different parameters and is defined as follows:

$$c^I.v := \frac{w_{st} * v_{st} + w_{ch} * v_{ch} + w_{int} * c_{int}}{w_{st} + w_{ch} + w_{int}}$$

where the w-values represent specific weights to adjust their corresponding v values. v_{st} is defined as follows:

$$v_{st} := \prod_{s \in c.S_C \cup c.S_E} v(s)$$

where $c.S_C \cup c.S_E$ is the set of all Statements belonging to $c^I.c$ and $v(s)$ returns the value of a Statement s. Each Statement is part of the current Construction instance c^I. The value therefore represents the product of all Statement results which were created for the instance c^I. The value states how good the Construction $c^I.c$ could be

applied to the specific context.
Next, v_{ch} is defined as

$$v_{ch} := \frac{1}{|con(c^I.Y^I)|} \sum_{y^I \in con(c^I.Y^I)} y^I.v * \frac{y^I.lvl}{c^I.lvl - 1}$$

where $con(c^I.Y^I)$ returns all instantiated ConstructionSymbols of $c^I.Y^I$, $y^I.v$ represents the likeliness value of the Construction instance referenced by the Symbol y^I and $y^I.lvl$ specifies the level of the Construction instance referenced by the Symbol y^I. The purpose of v_{ch} is to evaluate how good the referenced Construction instances fit. The assumption is that the better the 'foundation' of an instance is the higher its likeliness should be. The level of a Construction instance $c^I.lvl$ is computed as follows:

$$c^I.lvl := \begin{cases} 0 & \text{if } \#(c^I.ref) = 0 \\ MH(c^I.ref) + 1 & \text{if } \#(c^I.ref) > 0 \end{cases}$$

where $MH(c^I.ref)$ returns the maximum level of all referenced Construction instances. The level of an instance is always the maximum of all referenced Construction instance levels plus one. An illustration can be seen in figure 4.7. The instances i_1, i_2 and i_3 have the level 0, i_4 has the level 1 and i_3 the level 2.
The last value v_{int} is defined as follows:

$$v_{int} := \frac{|leafs(c^I)|}{width(c^I.int)}$$

where $leafs(c^I)$ returns a set containing all child Construction instances of c^I which do not have any children themselves, i.e., they are leafs; further $width(c^I.int)$ calculates the interval width which the Construction instance covers. v_{int} therefore gives an estimate of how well the Construction covers the area that it has been applied to. Ideally the value should be 1, i.e., each word within the interval of the Construction instance has also been referenced by the Construction. The lower v_{int} gets, the more unlikely the instance is suited to its specific sentence position because it is not capable of using all the information available within the interval.

All three values v_{int}, v_{ch} and v_{st} together specify the likelihood of c^I. Several experiments with varying values were performed for the different weightings, however, the best results were obtained when all three weighting variables had the same value.

During the process of evaluating EffectStatements it might happen that one Statement creates information which apply to a node of the syntax tree which has already been computed. This can be problematic as new information could lead to completely new information somewhere else. Therefore, the algorithm in 1 contains an additional mech-

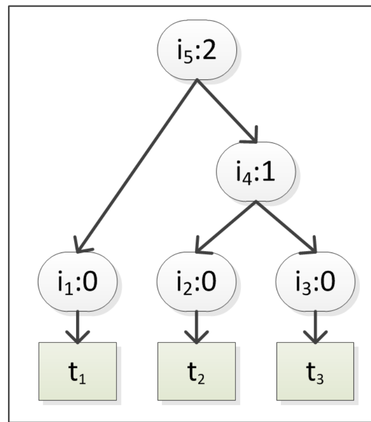


Figure 4.7.: Levels of different instances

anism which rechecks each node for changes. If it detects new information in a node, the process will restart at this node and again reapply all Constructions.
The process continues until no more Construction instances can be created. The state is reached if the Construction instance set C^I is complete:

Definition 22 (*Complete Construction Instance Set*)

C^I is called complete iff

$$\forall c \in O_c \ \forall s \in S_n \ \neg canCreateNewInstances(c, s)$$

where *canCreateNewInstances* is a predicate which returns true if for a given Construction c and a syntax tree node s new Construction instances can be created.

If the Construction instance is complete, the interpretation process for the current sentence is finished. Next, those Construction instances in C^I have to be selected which represent the root of a solution to the currently analyzed sentence:

Definition 23 (*Solution Construction Instance Set*)

Let

$$C_s^I \subseteq C^I \quad (4.18)$$

be a set of Construction instances which represent a potential solution to a sentence. In order for a Construction instance to be part of C_s^I it is not allowed to be referenced by other Construction instances from the same sentence, i.e., :

$$\forall x \in C_s^I \ \forall y \in C^I \ sameSen(y, x) \land \neg ref(y, x) \quad (4.19)$$

where $sameSen(y,x) \Leftrightarrow y.sen = x.sen$ and $ref(y,x)$ returns true if a directed reference from y to x exists. A Construction instance $c^I \in C_s^I$ is also called a solution.

4.2.3. Solution Extraction

So far, the Construction application created an internal data structure which is similar to a hypergraph [129]. The challenge with a hypergraph like structure is that it is 'compressed', i.e., a Construction instance which represents a specific semantic information exists only once even if the instance is part of multiple different solutions. Additionally, all solutions for every sentence in an analyzed text are also part of the same hypergraph structure. This basically means that the solutions have to be extracted from the hypergraph in two different dimensions.

The first dimension can be seen in figure 4.8. The figure shows an abstract representation of a sentence with 5 different words. Because there are 5 words there are also 5 different Construction instances (numbers 8 to 12). Each of these instances is referenced by one or multiple other instances (1 to 7). Some of those instances in figure 4.8 are being referenced by multiple other instances, i.e., instances 6, 8 and 10. If an instance is referenced by several other instances it is part of multiple solutions. The number of available solutions is specified by the number of elements within C_s^I (see definition 23), which contains the root elements of the different solutions (in figure 4.8 this is the upper 'Solutions' layer). In order to create the semantic interpretations for a single sentence the Construction instance trees for every $c_s^I \in C_s^I$ have to be extracted. For the example in figure 4.8 this leads to the solutions $\{1,5,8,9,10\}$, $\{2,8,6,10,11\}$, $\{3,6,10,11\}$ and $\{4,7,12\}$. Each of these four sets contains all the information necessary for an interpretation to be created.

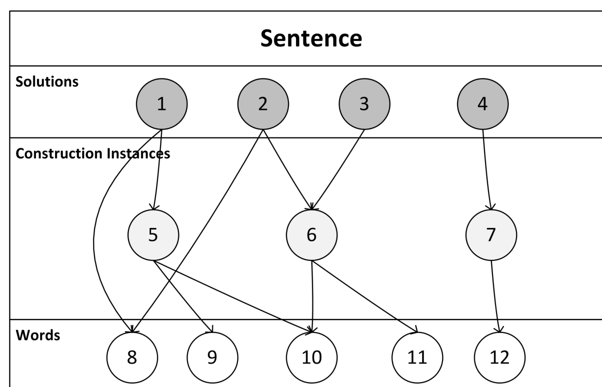


Figure 4.8.: Example of a hypergraph structure for a single sentence

The second dimension contains the connections between the different solutions of the single sentences which can be seen in figure 4.9. The figure shows three sentences and for

each sentence multiple solutions. Some of these solutions are connected to the solutions of other sentences. Others, however, do not have connections to either the previous (5, 7) or the following sentence (3).

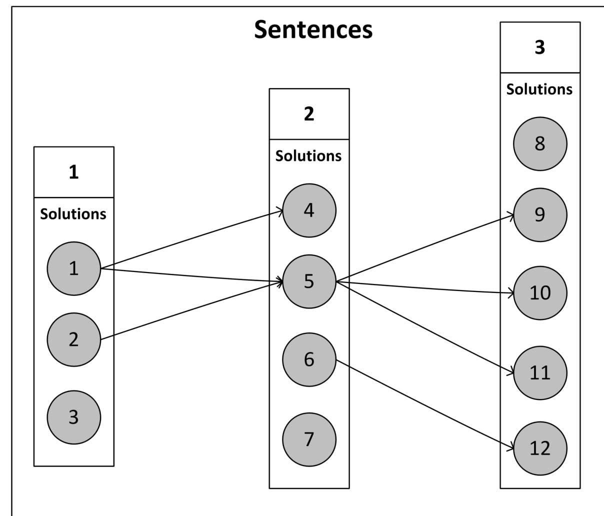


Figure 4.9.: Example of the hypergraph structure for multiple sentences

The goal is to create a set of so called solution paths:

Definition 24 (*Solution Path*)

Let

$$p := [c_0^I, .., c_n^I] \quad (4.20)$$

be a solution path consisting of an ordered list of solutions $c_0^I, .., c_n^I \in C_s^I$ with n being the number of parsed sentences n. Further, c_i^I is a solution for the (i+1)th sentence, e.g., c_0^I is a solution for the first sentence of a text. Further let

$$P := \{p_1, .., p_n\} \quad (4.21)$$

be the set containing all possible solution paths for a given text.

The term 'path' originates from connections between solutions in the hypergraph (as seen in figure 4.9). Hence, for every possible path, starting with a solution for the first sentence every possible path has to be created. In the example in figure 4.9, starting from node 1, one path is $\{1,5,9\}$, another one is $\{1,5,10\}$ and so on. A solution for a text consisting of one or more sentences ideally has a solution for every single sentence. If there is no solution for each sentence we call such a path fragmented. A solution fragment is specified as follows:

Definition 25 (*Solution Fragment*)

A solution fragment is defined as

$$F \subseteq C_s^I \tag{4.22}$$

where F only contains elements from C_s^I which are connected to each other:

$$\begin{aligned} isFragment(F) \Leftrightarrow \exists c_1^I, c_2^I, c_3^I, .., c_m^I \in F\, connected(c_1^I, c_2^I) \wedge connected(c_2^I, c_3^I) \\ \wedge .. \wedge connected(c_{m-1}^I, c_m^I) \end{aligned} \tag{4.23}$$

where $connected(c_x^I, c_y^I)$ indicates that the solution instances c_x^I and c_y^I are connected, i.e., there is not necessarily a direct relation between them but one of the instances within the solution tree of c_x^I must be connected to one instance of the solution tree of c_y^I or vice versa (e.g., this can be the case due to the pronominal anaphora resolution). The definition states that a fragment is a set of solution instances which are connected to each other. A path is called fragmented if there is no solution instance available for one or more sentences in the path or a solution instance c_l^I (with $1 \leq l < n$) is not connected to the solution of the following sentence c_{l+1}^I.

With the definition of a solution fragment we can specify an ideal path:

Definition 26 (*Ideal Solution Path*)

A path $p \in P$ is called ideal if

$$idealPath(p) \Leftrightarrow oneFragment(p) \wedge allSentencesSolved(p) \tag{4.24}$$

where $oneFragment(p)$ validates if the path contains only one fragment and $allSentencesSolved(p)$ checks if the number of solutions equals the number of sentences in the initial input text.

Therefore, if a path is fragmented (as it would be the case in the example $\{1,4,\}$), it represents an incomplete or not ideal solution. However, our aim is to always provide paths with n elements if the text contains n sentences. The algorithm therefore has to fill the gaps in fragmented paths if possible. For example, the path $\{1,4,\}$ could be filled with the instance 8, resulting in the path $\{1,4,8\}$. Another fragmented path would be $\{3,,\}$. Enriched versions of the path would be $\{3,7,8\}$ as well as $\{3,6,12\}$. In figure 4.9 no other paths are possible which would start with solution 3. The reason is that each fragment itself must be either complete or may not exist at all (fragments can not themselves be fragmented). This means that a path could not consist of $\{3,5,9\}$ because 5 has a dependency on 1 or 2. The restriction also implies that there can be cases in which

a path can not be fully enriched. In this case the path can not be considered any further and will be deleted.

As mentioned previously, all Construction instances are part of one hypergraph structure. In order to create solution paths, all fragments have to be identified first. The process starts with the elements within C_s^I. To create the different fragments the extraction process needs to find connections to the solutions of other sentences. This is done as shown in algorithm 3. The main method is *identifyFragments* and is called with an input parameter set of all solution instances. It first creates a hashtable $fragments$ (line 2) which stores all relations between all Construction instance solutions. Next, all elements in C_s^I are traversed. For each element $c^I \in C_s^I$ a list $frags$ is being generated. The list stores all solution instances of other sentences that c^I is connected with. The identifi-

Algorithm 3 Fragment Identification Algorithm

Input: C_s^I: Set of Construction instance solutions
Effect: Returns all identified fragments

```
1: procedure IDENTIFYFRAGMENTS(C_s^I)
2:     fragments := []
3:     for all ci in C_s^I do
4:         frags := {}
5:         collectConnections(ci, frags)
6:         fragments[ci] := frags
7:     end for
8:     return fragments
9: end procedure
```

cation of those solutions is done within the recursive method *collectConnections*$(c^I, conn)$ (see algorithm 4). The first parameter is the Construction instance for which the connections are collected. The second parameter is the list in which all related solution instances are stored. *collectConnections* goes through the set of relations c^I*.rel* that the current Construction instance possesses. For each relation $r^I \in c^I$*.rel* (line 2) it checks the target Construction instance $r^I.c_t^I$. If $r^I.c_t^I$ belongs to a different sentence (line 3), it calls the *getIsPartOfSolutions*$(r^I.c_t^I)$ procedure. This returns a subset of C_s^I of all Construction instances which themselves use the Construction instance $r^I.c_t^I$ as part of their solution (as we are still in a hypergraph structure, different solutions can still reference the same Construction instance). All Construction instances in the result of *getIsPartOfSolution* must be part of the same sentence as $r^I.c_t^I$. The result is added to the *conn* list. Next, the *collectConnection* procedure recursively checks all other Construction instances which are part of the solution and tries to identify all remaining relations to different sentences. As soon as *collectConnection* is finished, all the collected elements within *frags* are added to the *fragments* hash table (line 6 in algorithm 3). The process is repeated for each remaining element in C_s^I. At the end, *fragments* contains all fragments for every $c^I \in C_s^I$.

Algorithm 4 Connections Collection Algorithm

Input: c^I: A single Construction instance
conn: An initially empty set
Effect: Fills the *conn* list with all available connections to Construction instances of other sentences than c^I

1: **procedure** COLLECTCONNECTIONS(c^I, *conn*)
2: **for all** r^I in $c^I.rel$ **do**
3: **if** $r^I.c^I_t.sen \neq c^I.sen$ **then**
4: $conn := conn+$ getIsPartOfSolutions($r^I.c^I_t$)
5: **else**
6: collectConnections($r^I.c^I_t$, *conn*)
7: **end if**
8: **end for**
9: **for all** (**do**r in $c^I.ref$)
10: collectConnections(r, *conn*)
11: **end for**
12: **end procedure**

The result is used to create the solution paths. The process is explained in figure 4.9, which shows the different fragments. To create the solution paths we start with the first sentence. If the Construction instance c^I_1 in sentence 1 has a connection to the following sentence 2, represented by instance c^I_4, the connection will be followed and both will be added to a path p_1. However, there is no connection from c^I_4 to an instance solution of sentence 3. Hence, the algorithm analyzes all Construction instance solution of sentence 3 having no connection to sentence 2. As can be seen, there is c^I_8 which fulfills these condition and can therefore be added to p_1. The path p_1 is now complete and the algorithm continues with the next potential path p_2. It starts with c^I_1 and c^I_5. c^I_5 has multiple connections to the third sentence ($c^I_9, c^I_{10}, c^I 11$). Therefore, we have to clone p_2 for each of these connections. Since three connections to the third sentence exist, the paths p_{21}, p_{22} and p_{23} are created. Next c^I_9 is added to p_{21}, c^I_{10} to p_{22} and c^I_{11} to p_{22}. The algorithm continues until all potential paths have been generated.

The pseudo-code representation can be seen in algorithm 5. The main procedure is *CreateSolutionPaths* with input parameter *frags* (the fragments set of algorithm 3). The algorithm initializes the set P which stores all generated paths. Next, the procedure creates a set of all fragments starting with a solution for the first sentence (line 3) and storing it in variable $frags_t$. An iteration over each element of the set (line 4) creates a new path p for every fragment within $frags_t$. Next it calls the procedure *fillPath* (line 6) which tries to fill the remaining gaps of the path. It receives three different input parameters (algorithm 6): First, the path p which should be filled, the set of available fragments and the set P

Algorithm 5 Solution Path Creation Algorithm

Input: *frags*: A set of all available solution fragments
Effect: Creates and returns all possible solution paths P

```
1: procedure CREATESOLUTIONPATHS(frags)
2:     P := {}
3:     frags_t := getFirstSentenceFrags(frags)
4:     for all F in frags_t do
5:         p := new Path(F)
6:         fillPath(p, frags, P)
7:     end for
8:     return P
9: end procedure
```

which stores the newly created paths. It begins with validating if the path p is complete (line 2), i.e., if it contains a solution for every sentence or not. If p is complete, it is added to P (line 3). If it is not complete, the procedure tries to fill the gaps. Hence, it calls the method *findMatchingFragments* which searches the first gap within p and tries to identify fragments within *frags* which fit the gap (line 5). It traverses the result of the procedure (line 6), creates a clone of the path for each fragment (line 7) and adds the fragment f to the cloned path p_t (line 8). The path p_t may, however, still contain gaps, therefore the procedure calls itself (line 9) such that the remaining gaps can also be eliminated. The complete algorithm returns the set P (line 8 in algorithm 5).

Algorithm 6 Fill Solution Paths Algorithm

Input: p: A not necessarily complete solution path
frags: A set of all available solution fragments
P: The set of all final solution paths
Effect: Fills a path p with fragments from *frags* and adds it to P

```
 1: procedure FILLPATH(p, frags, P)
 2:     if complete(p) then
 3:         P := P + p
 4:     else
 5:         frags_m := findMatchingFragments(p, frags)
 6:         for all F in frags_m do
 7:             p_t := clone(p)
 8:             p_t := p_t + F
 9:             fillPath(p_t, frags, P)
10:         end for
11:     end if
12: end procedure
```

4.2.4. Creating the Interpretation Model

Based on the solution paths the InterpretationModels are created. These are contained within the InterpretationScope (see definitions 9 and 10). The code for creating the InterpretationModels is outlined in algorithm 7. It creates a separate InterpretationModel m for every single path $p \in P$. m is filled with the instances and all the information they reference (as seen in line 6; Note that all Construction instances are added to the set C_x^I). This is done by first adding all Construction instances which a solution instance $c^I \in p$ references (the attribute which contains the information is $c^I.ref$). A Construction instance c^I is represented as a ConstructionInterpretation $e \in m.E_{con}$. $e.con$ references the Construction c which is specified in $c^I.c$. Further, $e.sem$ points to the SemanticElement $c^I.sem$, i.e., a SemanticElementInterpretation e_{se} is inserted in $m.E_{sem}$, which references $c^I.sem \in O_{se}$. $e.sem$ is connected to e_{se} via an InterpretationRelation $r \in m.R_{int}$. This represents that e references the same SemanticElement as c^I does. Similarly the SyntacticElements within $c^I.syn$ are represented in $e.E_{syn}$.

Next, all Construction instance relations are inserted. Construction instance relations are transformed into AssociationInterpretations in the InterpretationModel. Therefore, for each Construction instance $c^I \in p$, each of its Construction instance relations r^I within either $c^I.rel$ or $c^I.e$ is added as an AssociationInterpretation to $e.R_{ass}$. Each reference within $c^I.ref$ is transformed to an InterpretationRelation which connects the corresponding ConstructionInterpretations.

Algorithm 7 Interpretation Models Creation Algorithm

Input: P: The set of all solution paths
Effect: Creates an InterpretationModel for each $p \in P$

```
 1: procedure CREATEINTERPRETATIONMODELS(P)
 2:     for all p in P do
 3:         m := createInterpretationModel(p)
 4:         C_x^I := {}
 5:         for all c^I in p do
 6:             addConstructionInstances(m, c^I, C_x^I)
 7:         end for
 8:         for all c^I in C_x^I do
 9:             addConstructionInstanceRelations(m, c^I)
10:         end for
11:     end for
12: end procedure
```

4.2.5. Detecting Termination Cases

Every algorithm has to terminate. This is simple for several cases but can become difficult in other ones. In the following we describe the termination conditions for the overall process.

Complete Construction Instance Set Basically the goal of the algorithm is to create every possible interpretation of a sentence. If no new information can be created, the Construction application phase will terminate. This is the case if C^I is complete (see definition 22).

Maximum Tree Height Constructions can be designed and modeled in a way such that they lead to an infinite loop in the application phase. The most trivial example can be seen in the previous section 3.5.4 in figure 3.13. Such simple cases can be detected. However, it must not necessarily be a single Construction which forms the loop but a set of Constructions or even Statements which introduce cycle. For example, a Construction $c_1 \in O_c$ is applied to a specific context. Now $c_2 \in O_c$ can apply to a context, which contains an instance of c_1. The information which c_2 creates can now again lead to instances which would match the Statements in c_1 and so on. A formal representation of this is given in definition 27.

Definition 27 (*Construction Loop*)

Let

$$C_t \in O_c \tag{4.25}$$

be a set of Constructions which can be applied to C^I. Let

$$application : O_c, \mathcal{P}(C^I) \rightarrow \{True, False\} \tag{4.26}$$

be a predicate where *application* instantiates a Construction from O_c on a set of Construction instances (see section 4.2.2) and returns *True* if the instantiation was successful. C_t contains a loop, if the following holds:

$$\begin{aligned} loop(C_t) \Leftrightarrow \exists c_1, c_2, .., c_j \in C_t \, application(c_1, C^I) \Rightarrow application(c_2, C^I) \Rightarrow \\ .. \Rightarrow application(c_j, C^I) \Rightarrow .. \Rightarrow application(c_1, C^I) \end{aligned} \tag{4.27}$$

This means that if a set contains different Constructions c_1 to c_j such that the application of c_1 allows the application of another Construction c_2, which again leads to the application of Construction c_1 the set is said to contain a Construction loop. Note

that c_1, c_2 etc. can actually be the same Construction.

The fact that an arbitrary number of Constructions could lead to such a situation makes a structural detection difficult and performance intensive. A better solution is based on threshold heuristics. As described previously, the overall process has been aligned to a syntax tree. The Construction instances belonging to a sentence form a tree like structure as one Construction instance references other Construction instances. Hence, the first threshold is based on comparing the syntax tree height to the tree size of the Construction instances. However, the syntax tree which was generated by the syntax tree parser might be either much higher or smaller than the Construction instance tree. This means that we need a heuristic which calculates the potential maximum height of a valid Construction instance tree. The value is oriented towards the size of the natural language text input W. Experiments showed the average number of children $T_{children}$ a node in the Construction instance tree has. Based on $T_{children}$ and the number of words in the current sentence an average size for a Construction tree is calculated:

$$maxCITreeSize(w) := \frac{log(w) * T}{\log(T_{children})} \tag{4.28}$$

with $w := \#(W)$, i.e., the number of words in the current sentence, $T_{children}$ the constant which represents the average number of children per node and $T \geq 1$ a constant weight which adds an additional safety buffer (the value of the parameter must be sufficiently large as a too small value could terminate the Construction application process before the correct result has been created). Then

$$\forall c^I \in C^I maxLevelReached(c^I) \rightarrow notUsable(ci) \tag{4.29}$$

where

$$maxLevelReached(c^I) = \begin{cases} true & \text{if } c^I.lvl >= maxCITreeSize(w) \\ false & \text{if } c^I.lvl < maxCITreeSize(w) \end{cases} \tag{4.30}$$

and $notUsable(c^I)$ marks a Construction instance as not usable, i.e., it can not be referenced by other Construction instances and does therefore not lead to new information. As soon as all $c^I \in C^I$ are marked with $notUsable$, no new Constructions can be applied, therefore C^I is complete and the Construction application process terminates, according to section 4.2.5.

Termination The tree height termination heuristic combined with the complete Construction instance set termination is sufficient to show that the Construction application process always terminates. We assume the following:

1. The height of a Construction instance $c^I.lvl \in C^I$ is specified according to definition 21. Therefore, the height of a Construction instance is always bigger than the one of its referenced instances (except for the case where a Construction instance does not reference any other instances).
2. The second assumption is that it is always known which Construction instances have been created. The information is stored within C^I_t (definition 18).

Based on these two assumptions we can show that the algorithm terminates. In the following, we look at different cases and show that all of them terminate.

1. The first case consists of a sentence in which no Constructions can be applied in the mapping phase. Hence, no Constructions exist in C^I. As no new Constructions can be instantiated (as they require some information to begin with), C^I does not change and is therefore complete. In this case, the heuristic from 4.2.5 leads to the termination of the Construction application process.
2. The second case consists of a (partially or fully) mapped sentence and therefore at least some information in C^I. Further we have a perfect set of Constructions O_c, i.e., $\neg loop(O_c)$ (the Constructions can not form a loop). In this case the system comes to a point where it can not deduce any new information with the given set of Constructions in O_c because, based on the assumption in 2, duplicate information can not be created (a duplicate Construction can be detected because of C^I_t). The process therefore terminates because no loop exists.
3. The third case consists of a (partially or fully) mapped sentence and therefore at least some information in C^I. We further have an imperfect set of Constructions O_c, i.e., $loop(O_c)$ (the Constructions can form a loop). Based on assumption 1, a Construction automatically increases its height as soon as it references other Constructions. Moreover, a new Construction instance always has a higher level than the instances it references. Therefore, if Constructions can form a loop, the corresponding Construction instances increase their level until the tree height termination heuristic marks the highest nodes as unusable (see section 4.2.5). Hence, no Constructions above this height can be created. The Construction application process can therefore only continue until

$$\forall c^I \in C^I \; maxLevelReached(c^I) \tag{4.31}$$

As soon as the state is reached, C^I is complete and the process again terminates.

The previous section described in detail the process of applying Constructions to a given sentence and the creation of an InterpretationModel. In the following, we describe how the complexity of the system can be handled.

4.2.6. Handling runtime complexity

As it was shown previously in section 4.2.2 a big challenge is handling the runtime complexity. The amount of potential new Construction instances equals

$$\#(elems)^{\#(conSymb(c.Y))} \tag{4.32}$$

where $\#(elems)$ is the size of the set which contains all available Construction instances for a given syntax tree node. Further, $\#(conSymb(c.Y))$ is the number of Construction-Symbols of the to be applied Construction c. Complexity can be handled as follows:

1. Pruning Construction instances from *elem*: Prior to actually instantiating the Constructions, algorithm 2 creates a Cartesian product in line 4. The function internally creates $\#(conSymb(c.Y))$ copies of the *elems* list. Each of these lists can be filtered according to the Functions which have been specified in the current Construction c. The most important Functions are described in section 4.3. Most interesting in this context are the Functions 'InOrder' in section 4.3.1 and 'IsOfType' in section 4.3.4.
2. Limiting the amount of newly created Construction instances: We designed an algorithm which takes all newly created Construction instances of a syntax tree node and filters them. Therefore, the new instances are first grouped according to the Construction they are based on. Only those which have a good interval ratio (see the definition of v_{int} in definition 21) and a high likeliness value remain within C^I.
3. Construction design: Finding a good trade-off between Construction tolerance, precision and parsing performance is a difficult task. We focus on this part later in the evaluation chapter 7, especially in both case studies.

Those mechanisms greatly reduce the overall amount of instances and allow the parsing of a natural language text within reasonable time.

4.3. Functions

In this section we describe a set of Functions which are currently being used for defining Constructions and therefore identifying syntactic structures. The Functions have been designed to be usable in the German as well as English language. Every Function has two ways of providing functionality for the parsing process. The first method is called during initialization as it was explained in section 4.2.6, and can eliminate elements from the different copies of the *elems* lists. The second method is called during the instantiation process of a Construction. Each Function returns a value in the range [0..1]. The following sections explain the concepts behind these Functions and show how the return value is being computed.

4.3.1. Correct Word Order

In many languages the word ordering is important for the meaning of a sentence. It is therefore necessary to identify a specific order of words by using the ConditionStatements within Constructions. We define the correct order of words as follows:

Definition 28 (*Correct Word Order*)

Let

$$W_t := (w_a, .., w_n, .., w_m) \tag{4.33}$$

be a list whose elements have randomly been chosen from the words of the natural language input text, i.e., $w_a, w_n, w_m \in W$. The words within W_t are in the wrong order, if

$$\begin{aligned} wrongOrder(W_t, W) :=& \exists w, w_n \in W_t \; notEqual(w, w_n) \\ & \wedge isWrongOrder(w_n, w, W, W_t) \end{aligned} \tag{4.34}$$

where $notEqual(w, w_n) \Leftrightarrow w = w_n$ checks if two words are identical within W_t. $isWrongOrder(w_n, w, W, W_t)$ validates if two words w_n and w of the list W_t are in the wrong order. The order is specified with respect to the original text W, i.e., if w comes before w_n in W, it must also come before w_n in W_t. The predicate is true if the two given words are in the wrong order within W_t. The correct word order is the negation of the predicate *wrongOrder*.

$$correctOrder(W_t, W) \Leftrightarrow \neg wrongOrder(W_t, W) \tag{4.35}$$

An example can be seen in the following lists $W_t := \{w_3, w_1\}$ and $W := \{w_1, w_2, w_3, w_4\}$. W contains the words in their correct order. W_t contains two words from W, which are not in the correct order. W_t does not satisfy the *correctOrder* predicate since, with respect to W, w_3 comes after w_1.

The corresponding Function within SE-DSNL is called 'InOrder'. It takes an arbitrary number $n \geq 2$ of Symbols. The order of the symbols has to be checked against the initial input W (as defined in section 4.2.2) similar to the definition 28. The task, however, becomes more difficult if a Construction instance not only covers a single word, but a complete phrase. Phrases can be detected through the interval attribute $c^I.int$. An example can be seen in figure 4.5. Construction instances which have been created at syntax tree node N_3 contain the interval value [3..3] (or just short [3]), whereas Construction instances at node S_2 could have an interval value [3..5] because those might cover the whole range of tokens from N_3 to N_4. Note that an interval [3..5] does not mean that necessarily all words in the interval are being referenced by the instance. Only words at position [3] and [5] are definitely part of the interval, the word [4] may be part of it but does not have to be. This depends on the Construction and the information it requires. In the example in figure 4.5, one Construction would interpret the verb V_2 as a one-argument structure and hence only require the noun N_3. Another Construction, however, would identify the complete subject-predicate-object structure and therefore make use of N_3, V_2 and N_4.

In the following, we consider int_1 and int_2 to be two intervals. As shown before, intervals can either point to a single position in W (as it is the case with single words) or reference a whole subsection of W. Four different cases have to be considered:

1. The intervals overlap, i.e., $int_1 \cap int_2 \neq []$. This happens in a case where the nodes of the syntax tree are overlapping. An example for this can be seen in figure 4.10. One Construction instance has an interval $int_1 := [0..3]$ and the other Construction instance $int_2 := [2..5]$. Let $intersection := int_1.end - int_2.start$ and $covereddistance := int_2.end - int_1.start$. The result is defined as $result := 1 - \frac{intersection}{covereddistance}$. The idea is that certain parts of the intervals are not exactly in order, therefore those overlapping parts have to be weighted against the rest. In our example the result is $result := 1 - \frac{1}{3}$.

2. The second case is an interval which lies completely within another interval. An example is $int_1 := [1..6]$ and $int_2 := [2..4]$ or $int_2 := [5]$. Both times, $int_2 \subseteq int_1$. The problem with a total overlap is that there can be no order identified. Hence, the result is $result := 0$.

3. The third case is defined as both intervals being in the correct order, i.e., $int_1.end < int_2.start$. This means that the end of int_1 lies before the beginning of int_2. To calculate if the order is correct the gap between both intervals is measured, i.e., $gap := MIN(int_2.start - int_1.end; DIST_{max})$, where $MIN(..)$ returns the smallest value

and $DIST_{max}$ is a constant which specifies the maximum gap which should exist between two intervals. The result is defined as $result := \frac{DIST_{max}-(gap-1)}{DIST_{max}}$. As can be seen, we integrated the distance between both words in the calculation of the result value. It means that the smaller the gap between two intervals is, the better.

4. The fourth case is specified by both intervals being in a completely different order. As humans make mistakes in writing we introduced a small tolerance. Therefore, $gap := MIN(int_1.start - int_2.end; WRONGDIST_{max})$, where $WRONGDIST_{max}$ is a constant which indicates the maximum allowed distance in case that the intervals are in the wrong order. Ideally $WRONGDIST_{max} < DIST_{max}$ as the value for wrong order should drop faster than for the correct order of words. The result is defined as follows: $result := \frac{WRONGDIST_{max}-gap}{WRONGDIST_{max}}$. In contrast to the correct order each wrong order automatically has a result smaller than 1. If, e.g., $int_2 := [3]$, $int_1 := [4]$ and $WRONGDIST_{max} := 2$, then $result := \frac{2-(4-3)}{WRONGDIST_{max}}$ which leads to $result := \frac{1}{2}$.

Based on these four cases the order of two intervals is calculated and a value in the range from [0..1] is returned.

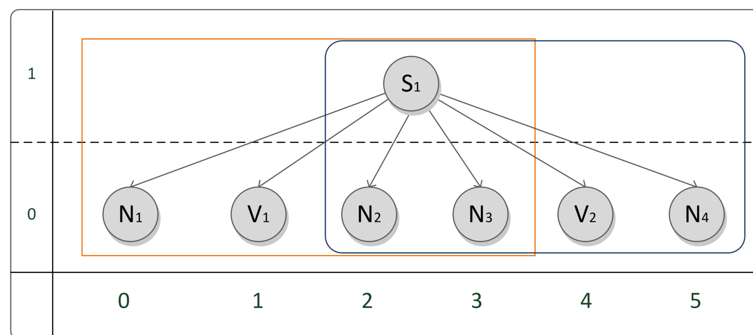


Figure 4.10.: Syntax tree with two overlapping Constructions instances; The blue colored instance additionally references node N_2, whereas the other instance would reference N_3

We previously described which constraints the 'InOrder' Function checks during Construction instantiation. However, the Function also provides mechanisms for eliminating Construction instances during the initiation phase. During the initiation phase, a list *elems* is collected (see line 3 in algorithm 2). The list is cloned according to the number of ConstructionSymbols of the current Construction $c.Y$, such that the Cartesian product can be built (this is required to check every possible linguistic combination such that SE-DSNL can be used with every possible language). If the 'InOrder' Function is called with the ConstructionSymbols cs_1, cs_2 and cs_3, three clones of the *elems* list are created, i.e., $elems_1$, $elems_2$ and $elems_3$. cs_1 are later instantiated with an element from $elems_1$, cs_2 with an element from $elems_2$ and cs_3 with one from $elems_3$. However, 'InOrder' implies that the textual representation of the element, used to instantiate $cs1$, must appear before

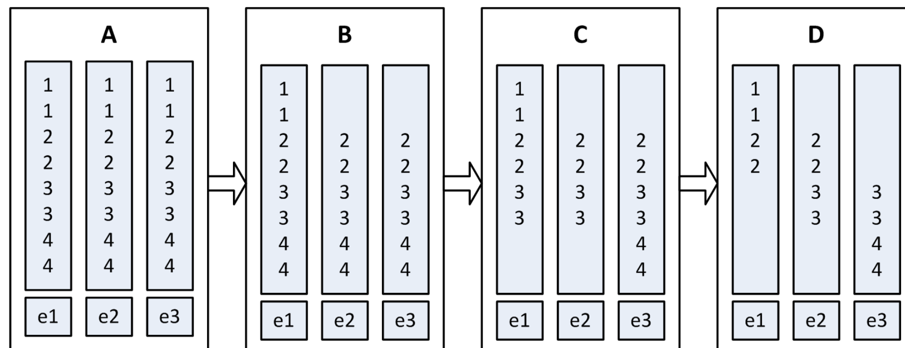


Figure 4.11.: 'InOrder' Function optimizing the *elems* list copies based on the position of the different entries

the textual represention of the element, which has been selected to instantiate $cs2$. The assumption can be used to delete elements from these three lists prior to creating the Cartesian product. Hence, 'InOrder' was extended with an algorithm which allows to filter the initial lists of the Cartesian product.

The filter-algorithm is based on the following assumption: Let $e_1..e_m..e_n$ be n equal lists containing Construction instances. Given a list of ConstructionSymbols $cs_1..cs_n$, each ConstructionSymbol must be instantiated with an element from its according list, i.e., cs_1 with an element from e_1 and cs_n with an element from e_n accordingly. Therefore, the element lists have to be in the same order as the ConstructionSymbols. This means that the elements in e_1 must contain the elements representing the leftmost textual elements, whereas e_3 must contain elements representing the rightmost textual elements. We define the following condition: For the element $i \in e_m$ (where $1 < m <= n$), which represents the leftmost textual word within e_m, there must be an element $j \in e_{m-1}$ which represents a textual element to the left of i. Additionally, if $i \in e_m$ (where $1 <= m < n$) is the element, which represents the rightmost textual element within e_m, there must be an element $j \in e_{m+1}$ which succeeds the rightmost element i within the text.

Let us demonstrate this fact by an example. Figure 4.11 shows four steps (A, B, C and D) of three *elems* list $e1$, $e2$ and $e3$, which have to be optimized. Each row represents one Construction instance and the number indicates the position of the textual element of the instance in the original input sentence. In step A the initial unfiltered state of the three lists is shown. The process starts by selecting the element which is leftmost in the sentence. In this case these are obviously the elements at position 1. Now the following lists are checked. As we know that each element in $e2$ and $e3$ must have an element before them in $e1$, list $e2$ and $e3$ are not allowed to contain elements at position 1. The reason is that the leftmost position in the text is position 1. If elements at position 1 would be allowed in list $e2$, no elements from $e1$ could be used to instantiate cs_1 and the instantiation would fail as soon as the normal 'InOrder' constraints would be checked. This case

is eliminated by deleting all elements at position 1 from lists $e2$ and $e3$. The result can be seen in step B. Next, the same is done starting from the last list, i.e., beginning with $e3$ the algorithm searches the elements at the rightmost position in $e3$, which are the elements at position 5. Following, the algorithm checks if elements at the same position are contained within $e2$ and $e1$. As can be seen, there are elements at that position, therefore these elements at position 5 are deleted from $e2$ and $e1$. The result is shown in step C. The algorithm now restarts with the first list $e1$ and again selects the leftmost element which is compared to the other lists. However, no elements from $e2$ or $e3$ violate the conditions. Hence, the algorithm selects the second leftmost position 2 from $e1$ and again compares the other elements to it. As can be seen, in list $e2$ the condition is not violated: For the elements at position 2 in $e2$ there are elements at position 1 available in $e1$. The same is true for elements at position 3. However, the elements at position 2 in list $e3$ violate the condition because the leftmost elements in list $e2$ are the ones position 2. Therefore, all elements with position 2 are deleted from $e3$. Next, the algorithm restarts with list $e3$ and works its way to the beginning. The only elements which violate the condition are the elements at position 3 in list $e1$. These are therefore deleted. Step D presents the final result of the algorithm, in which the remaining elements of all lists fulfill the conditions. The amount of elements in this case was reduced by 50%. This leads to a reduction of complexity from $8^3 = 512$ to $4^3 = 64$, which is $\frac{1}{8}$ of the original Cartesian product combinations. It is obvious that this greatly increases the overall performance.

4.3.2. Pronominal Anaphora Resolution

The detection of pronominal anaphoras is a more difficult problem than the calculation if two words are in the right order. Anaphora detection today is often based on machine learning approaches and the usage of several different linguistic features (as has been described in section 2.1.6.4). However, ontologies and conceptual knowledge is rarely incorporated into the process. The advantage of our approach is that there is a profound semantic knowledge base available. The semantic information should therefore also be usable for anaphora resolution. The overall assumption is that a verb references the pronoun which should be resolved. The semantic meaning of the verb is most likely in relation with the meaning of the antecedent, i.e., the noun which the pronoun refers to. An example can be seen in figure 4.12. There, the pronoun 'who' refers to the antecedent 'driver' which can sleep. We therefore base our pronominal anaphora resolution concept on the following assumptions:

Assumption 2 (*Pronominal Anaphora Resolution*)

If

1. A sentence contains a pronoun,

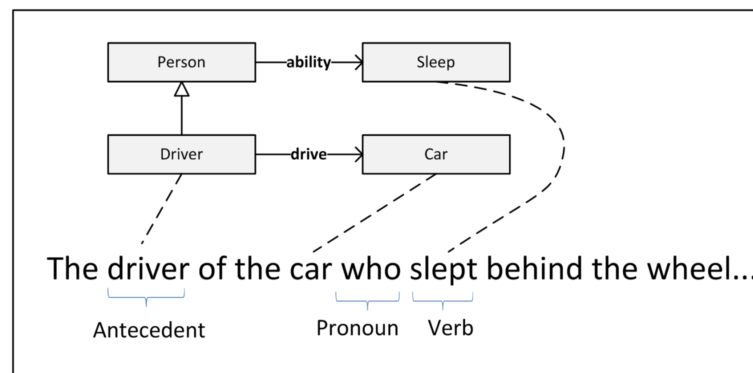


Figure 4.12.: Pronominal Anaphor Resolution Concept Overview

2. A noun which represents the antecedent of the pronoun is available within the sentence,
3. A verb directly refers to the pronoun and uses it as a substantive,
4. The semantic meaning of the words are known and related within the ontology,

it should be possible to identify the antecedent of the pronoun based on the semantic knowledge within a SE-DSNL model.

Theoretically there are many different forms of anaphora resolution, however, we simplified the challenge to resolving pronouns. A main motivation behind this idea was to only use the knowledge within a SE-DSNL model and not rely on additional databases and linguistic features like agreemenet (which is the case with standard anaphora resolution algorithms). The reason is that additional information sources would require additional well defined mappings to the SE-DSNL model.
Aside from the standard models for anaphora resolution like first order probabilistic models, (un-)supervised machine learning approaches (e.g., with Bayesian Models) as well as linear integer programming there are still more simplified approaches which, based on pair wise comparison and simplified distance measurement also yield good results in certain cases. Our approach is based on the idea of Bengtson and Roth [130]. They selected a simplified set of features together with a syntax tree based distance measurement. These features together yielded results that are comparable to that of more advanced machine learning based approaches.

In our approach we make use of only two features. The first one is a syntax tree based distance measurement. The second feature makes use of the semantic meaning of words with regard to the SemanticScope.

The syntax tree based distance measurement is explained first. An example of an abstract syntax tree can be seen in figure 4.13. The figure actually shows the syntax trees for two different sentences S_1 and S_2 as indicated by the corresponding root nodes. Simplistic approaches to measuring the distance between two words would use the information about the position of the words in the sentence. The syntactic distance d_{syn} between the nodes A and E would therefore be 4 (E is at position 4, A is at position 0, the result of $E - A$ is therefore 4). The definition of the syntactic distance is, however, not elaborated enough. One challenge is for example represented by the nodes C and D which form a constituent (indicated by node G). It would therefore be helpful if C and D would not count as two but only as one node. Constituents can build parts within a sentence that introduce additional information but are not necessary for the core meaning of the sentence. An example for such a syntactic structure would be a dependent clause. The first word after a dependent clause can directly be related to the words before the dependent clause. A naive distance measurement like the previous one would count all the words within the dependent clause even if the clause is not part of the main sentence.

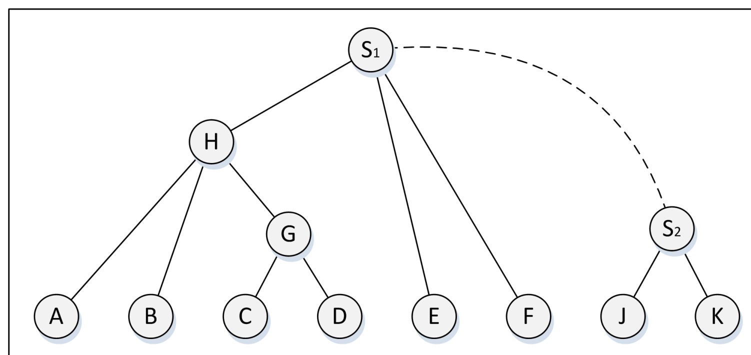


Figure 4.13.: Two Syntax Trees as an Example for Distance Measurement in Anaphor Resolution

The distance measurement must handle such obstacles. This means for the nodes A and E in figure 4.13 that their syntactic distance d_{syn} is only 3 instead of 4. Our algorithm is based on counting the number of direct sisters of each node in the direction to the other node. Let us consider an example. Starting from node A, its direct parent is H. H is not the Lowest Common Ancestor (LCA) of A and E (i.e., the first node within a tree that subsumes both A and E). Hence, the algorithm needs to move further upwards the tree in order to find the LCA node. However, before moving up the tree, it first counts the siblings to the right of A at node H, which increases the result by 2 (i.e., B and G). The algorithm analyzes the parent of H which is S_1. S_1 is the LCA of A and E and has two direct children, to the left H and to the right E. This increases the value of d_{syn} by one (the minimum distance between two nodes is always 1, e.g., C and D). The algorithm finishes at S_1 because it found the node E.

A specific challenge is that the antecedents are not always contained within the same but in a previous sentence. In order to parse several sentences we extended the algorithm: To calculate the distance between the nodes of multiple syntactic trees we connected the root nodes of both trees (as shown in figure 4.13 where a dotted line connects the nodes S_1 and S_2). As an example the algorithm measures the distance between the nodes J and E. It starts from J and moves up to the parent node S_2. S_2 is not related to E, the algorithm therefore continues its search by checking the root node of the previous sentence S_1. There it finds the node E, which is a direct child of S_1. Experiments showed that connections between the root nodes of different sentences should be rated slightly higher, i.e., in case that the searched node (here E) is part of a different sentence, we add a value higher than 1 to the distance result d_{syn} (a value of 2 has proven to be sufficient). Finally, the algorithm analyzes the siblings to the right of node E where it finds the node F. The node is located exactly between E and J. Therefore, the result is incremented by one. This leads to a distance value $d_{syn} = 3$ between the nodes E and J.

The simplified code is shown in algorithm 8 (the algorithm does not incorporate the functionality to calculate the distance over different syntax trees which was left out for clarity). The algorithm consists of basically three different procedures. The main procedure

Algorithm 8 Syntax Tree based Distance Measurement Algorithm

Input: n_{from}: The syntax tree node from which the algorithm starts searching for n_{to}
n_{to}: The syntax tree node which the algorithm is searching for, starting from n_{to}
Effect: Returns the distance between n_{from} and n_{to}

```
1: procedure COMPUTENODEDISTANCE(n_from, n_to)
2:     d_syn := 0
3:     lca := getLCA(n_from, n_to)
4:     d_from :=countNodesToLCA(n_from, n_to, n_from.parent, lca)
5:     d_to :=countNodesToLCA(n_to, n_from, n_to.parent, lca)
6:     d_syn := d_from + d_to+ countDist(n_from, n_to, lca)
7:     return d_syn
8: end procedure
```

ComputeNodeDistance gets two input parameters $n_{from}, n_{to} \in S_n$. It begins with initializing a variable d_{syn} which stores the final distance value. Next, the method *getLCA* is called which computes the LCA node of n_{from} and n_{to} (the procedure is not shown specifically but its functionality is to recursively walk up the syntax tree S and check each node if it is the LCA of two given nodes). Afterwards, the algorithm calls the procedure *countNodesToLCA* (line 4) with the parameters n_{from}, n_{to}, the parent of n_{from} and the *lca* node. The procedure (algorithm 9) counts the number of nodes which are in the tree from n_{from} up to the *lca* with respect to n_{to}, i.e., if n_{to} comes to the left or to the right of n_{from} within the syntax tree. The same is done for the other node n_{to} (line 5 in algorithm 8). Finally, the different distance values are summed up. Additionally, the distance between

n_{from} and n_{to} in the children list of the node *lca* is added to d_{syn}. The value is calculated in the method *countDist* (see algorithm 10). The procedure finally returns d_{syn} in line 7.

Algorithm 9 Count Nodes to LCA Algorithm

Input: n_1: The first node the algorithm searches for
n_2: The second node the algorithm searches for
n: The node the algorithm currently analyzes
lca: The LCA syntax tree node of n_1 and n_2
Effect: Counts the number of nodes from n_1 to n_2 and returns it in variable d

```
1: procedure COUNTNODESTOLCA(n1, n2, n, lca)
2:     d := 0
3:     if n ≠ null && n ≠ lca then
4:         d := countDist(n1, n2, n)
5:         d = d+ countNodesToLCA(n1, n2, n.parent, lca)
6:     end if
7:     return d
8: end procedure
```

Algorithm 10 Syntax Tree based Distance Measurement Algorithm

Input: n_1: The first syntax tree node the algorithm searches for
n_2: The second syntax tree node the algorithm searches for
n: The LCA syntax tree node of n_1 and n_2
Effect: Returns the distance d of the two child nodes n_1 and n_2

```
1: procedure COUNTDIST(n1, n2, n)
2:     d := 0
3:     ind1 := getIndex(n1, n.children)
4:     ind2 := getIndex(n2, n.children)
5:     if ind1 < 0 && ind2 ≥ 0 && n1.pos < n2.pos then
6:         d := ind2
7:     end if
8:     if ind1 < 0 && ind2 ≥ 0 && n1.pos > n2.pos then
9:         d := n.children.size − ind2 − 1
10:    end if
11:    if ind2 < 0 && ind1 ≥ 0 && n1.pos < n2.pos then
12:        d := ind1
13:    end if
14:    if ind2 < 0 && ind1 ≥ 0 && n1.pos > n2.pos then
15:        d := n.children.size − ind1 − 1
16:    end if
17:    if ind1 ≥ 0 && ind2 ≥ 0 then
18:        d := MAX(ind1, ind2) − MIN(ind1, ind2)
19:    end if
20:    return d
21: end procedure
```

The procedure is *countNodesToLCA* recursively counts the number of nodes between two nodes n_1 and n_2 up to the node *lca*. This is done by first calling the procedure *countDist* (line 12). Next, it recursively calls itself by going up one level in the syntax tree towards the *lca*. The procedure terminates as soon as the parameter n is either null or $n = lca$.

The procedure *countDist* calculates the actual distance between two nodes n_1 and n_2 on a given node n. It requires the index of the nodes within the list of children of node n (lines 19 and 20). In an ideal case the node n is the LCA of both n_1 and n_2. In this case the distance between both nodes can be calculated as shown in line 34, i.e., the difference between the index of n_1 and n_2 is the value which is returned. For example, in figure 4.13 the position of node B with respect to node H would be 1 and for node G it would be 2, therefore the distance between both of them is 1 (note that this is just a simplified representation of the actual algorithm because the procedure *getIndex* might receive the node C at node H and would still return the index 2). However, there are cases where n is not the LCA of both n_1 and n_2. Those cases are handled in lines 21 to 32. If for example node n_1 could not be found in the children list of n, its index ind_1 is set to -1. The return value depends on if the position of n_1 in the syntax tree is to the left or to the right of n_2. If n_1 comes to the left of n_2 (line 21), the procedure returns the index of n_2 (line 22). This is exemplified in figure 4.13. The procedure *countDist* is called with the parameters B, D and G thus it calculates the distance between nodes B and D on node G. However, B is not part of the children list of G. Hence, the return value is the index of D in the children list of node G,i.e., 1. Alternatively n_1 is placed to the right of n_2 (line 24). Then the return value depends on the size of the children list, from which the index of n_2 is subtracted. The cases for handling a missing node n_2 are similar to the previous ones and can be seen in lines 27 to 32.

The concept of calculating the distance is one of two features which helps to find the antecedent of an anaphoric pronoun. The other feature is based upon the semantic information within O_{se}, i.e., how close the SemanticElement, which represents the verb, is actually related to the SemanticElement of its potential antecedent. To calculate the semantic relatedness two different approaches are used:

1. The first one is based on the spreading activation algorithm, which is explained in section 5. Its input parameters are the SemanticElement of the potential antecedent as well as the SemanticElement of the verb which is related to the pronoun. The reason for selecting those two is that the meaning of the verb should be related to the meaning of the antecedent. This can be seen in figure 4.12. The spreading activation is initialized with the SemanticElements 'Driver' and 'Sleep'. It returns a value between 0 and 1 which indicates how close the SemanticElements are connected semantically (the higher the value the more likely a strong semantic relation exists between both elements). The value is stored in the variable d_{spr}

2. The second approach is based on a simple semantic distance measurement between the SemanticElement of the verb and the SemanticElement of the potential antecedent in O_{se}. The ontology is therefore treated as a normal graph on which a Dijkstra-based algorithm calculates the distance between both nodes by using Association and Generalization relations only. To subsequently use the value it has to be normalized to a value between 0 and 1 which is done as follows:

$$d_{simple} := 1 - \frac{MIN(d_{dijk}, d_{MAX})}{d_{MAX}} \tag{4.36}$$

where d_{dijk} is the result of the Dijkstra distance measurement algorithm, $MIN(d_{dijk}, d_{MAX})$ selects the smaller value out of d_{dijk} and d_{MAX} and d_{MAX} is a constant which represents the maximum valid distance value.

The values of both algorithms are combined using a simple arithmetic average:

$$d_{sem} := \frac{d_{simple} + d_{spr}}{2} \tag{4.37}$$

The advantage of using two different algorithms is that both perform well in different situations. The spreading activation based algorithm produces better results regarding the exact semantic relation between two concepts, whereas the Dijkstra based algorithm offers more tolerance to arbitrary semantic structures without caring about the actual semantic validity of the structure. The latter helps especially when certain information are not directly available within O_{se}.
For every antecedent, both the semantic relatedness d_{sem} as well as the syntactic distance d_{syn} to the verb is calculated. However, only those candidates whose syntactic distance d_{syn} is below a certain threshold and whose d_{sem} value is above a specific threshold lead to a new Construction instance in the set C^I. This can also mean that multiple new Construction instances are created as part of the process. The reason for adding multiple new instances is that although a preselection, regarding the antecedents, has been made it is not certain which of those instances is the best candidate with respect to the final solution.

The actual representation of a resolved anaphora within SE-DSNL is explained in the following. In order to reference the Construction instance i_r which represents the antecedent, the Construction instance i_p (which represents the pronoun) is cloned, resulting in i'_p. Next $i'_p.sem := i_r.sem$, i.e., the SemanticElement of i'_p is set to the SemanticElement of i_r. Further, an Association a between both i'_p and i_r is created. The Association represents that i'_p equals i_r, i.e., both actually refer to the same entity. Therefore, the type of the Association a has to be set to a SemanticElement which in the current SE-DSNL model represents the meaning of equality (e.g., in our test scenarios we simply created a

SemanticElement 'Equal' which was used for this case).

4.3.3. Check and Create Triple

Using background knowledge within the parsing process can be relevant in order to disambiguate word senses or to connect the correct concepts. To reach the aim a Function had to be defined which allows the access on the SemanticScope during the parsing process itself. It should return zero if no information about a relation between a given list of SemanticElements (containing three different SemanticElements) is available. The idea is that these three SemanticElements ideally form a triple. Now the Function should check if a triple of this kind exists within O_{se}. This is done by using the spreading activation algorithm which is described in section 5. The algorithm returns 0 if it cannot find such a triple within the ontology. It returns 1 if it can find the triple. In certain cases, however, the spreading activation algorithm identifies information (e.g., if some words were underspecified as specified in section 2.1.4) which are worthwhile to be added to C^I, e.g., for a word "Human" which is associated with the SemanticElement 'Person' only, the spreading activation algorithm might detect that instead of 'Person' the SemanticElement 'Driver' might be better suited. This is new information which represents an alternative to the initial and already existing Construction instance. Such alternatives can be added by the Function 'CreateTriple' which can be used by EffectStatements. The reason is that EffectStatements are only executed if all ConditionStatements of the same Construction have been evaluated successfully. Otherwise, too many irrelevant or even false Construction instances might be added, thereby decreasing precision and performance.

It could be argued that the Function may influence the outcome of the process for the worse, i.e., by checking the 'static' knowledge within the ontology it could introduce a bias into the analysis process. Bias means that the InterpretationModel should reference only what the author of the text originally meant. The analysis process tries to achieve this by using Constructions which should identify the correct syntactic and semantic structures. These structures can be checked with ConditionStatements. One ConditionStatement can, e.g., be used to validate if the SemanticScope contains information about the relations between a set of SemanticElements. The ConditionStatement returns a value which indicates if a relation between those SemanticElements could be found, i.e., it returns 0 if no relation could be found or 1 if a strong semantic connection is available. In the first case the problem is that the knowledge which would relate the set of SemanticElements in O_{se} may not yet have been created within the SE-DSNL model. Hence, the analysis process assigns the ConstructionInstance a lower value than it actually should be. In the other case (i.e., if the algorithm returns 1) it might be the case that the author of the text intended something else but the analysis process found a combination of

ConstructionInstances which accidentally matches the knowledge within O_{se}. Both cases could lead to results which are not intended by the author of the text.

However the problem might not be too relevant. The reason is that there are more things to be checked regarding the text than just the semantic meaning and its relations within O_{se}. A single Construction almost always consists of more than just one ConditionStatement. Therefore, the 'CheckTriple' ConditionStatement has only a small effect on the overall outcome of the ConditionStatements. Further, it is mainly used to rate different alternatives for one and the same situation, e.g., a word which has multiple meanings. The reason is that the Statements which check the syntactic content of the sentence have already ruled out many different possibilities. What remains is a set of potentially correct solutions for the same words. In such situations all syntactic Statements yield nearly the same result values, therefore it is the semantic Statements which can make the difference. This means that although a small bias might be introduced it most probably does not influence the complete meaning of the interpretation. It can therefore be neglected.

4.3.4. Semantic Type Check

The Function 'IsOfType' only provides functionality for the initiation phase, i.e., it eliminates not matching Construction instances from the *elems* list. Its task is to verify that a Construction instance contains a specific type. This is helpful to identify if a Construction instance either contains a specific POS type or a SemanticElement. 'IsOfType' has therefore been designed to receive a reference on a Construction instance $c^I \in C^I$ as well as a SyntacticSymbol or SemanticSymbol s. With these arguments it checks if $c^I.sem \subseteq s$ or $s \in c^I.syn$. If this is the case, the Function returns 1, else 0.

4.4. Related Work

Many concepts have been proposed in the past which require a combination of linguistic and semantic information. In the following sections, we especially present approaches from two domains, which elicit similarities to SE-DSNL: OBIR and OBIE. Both have to cope with identifying semantic information in text. Various approaches are presented and delimited from SE-DSNL, before some concepts which mainly focus on semantic annotation, are described. It should be pointed out that most of those systems are intended for annotating large sets of documents, whereas SE-DSNL focuses on how the precision in handling language in an OBIR system can be tackled. We will therefore focus on such systems and delimit them based on how well they make use of available ontological information for linguistic processing.

We also want to point out that there are many other systems which apply ontological information to NLP related tasks like WSD (e.g., [131] [132] [79] [133] [134]). However, those systems focus on the analysis of a specific linguistic challenge only. In contrast, the focus of SE-DSNL is to provide an OBIR system which can be adapted to an arbitrary domain and its (linguistic) challenges, thereby enabling a better mapping between language and semantic knowledge. This goal is supported by the analysis of the SemanticScope with Functions (section 4.3). In contrast, many of the available OBIR and OBIE systems do not seem to make much use of the available semantic information, at least not during analysis runtime (many systems only use ontological information to validate the results in the end). Therefore, we delimit SE-DSNL from those by showing whether they analyze challenges like underspecification / WSD with the help of ontological knowledge or not.

The section is concluded in subsection 4.4.4.

4.4.1. Ontology-based Information Retrieval

OBIR systems are known under many different names, the most prominent ones being semantic information retrieval or concept based information retrieval. Their task is to retrieve information from a specific type of sources (e.g., natural language text). These systems try to identify semantic information from documents and retrieve documents based on the semantic knowledge. The section only copes with the process of identifying the semantics within text. Most OBIR systems create an index which contains semantic information of all searchable documents. The InterpretationModel of SE-DSNL can be seen as a semantic index of an input text and the SemanticScope is comparable to what other approaches consider being their ontology. In the following different systems from this research area are presented.

Vallet et al. [135] developed a system which allows the "semi-automatic annotation of documents" as well as a retrieval model. Similar to the SE-DSNL concept, their approach requires the ontology to conform to certain guidelines, e.g., specific different base classes, which built the center of different taxonomies. Their annotation and extraction system is built upon the system by Kiryakov et al. [136] and makes use of GATE [137]. GATE provides different modules for many different linguistic challenges, e.g., named entity recognition, pronominal anaphora resolution etc.. It is, however, unclear if Vallet et al. make use of components for anaphora resolution. At the center of their pipeline lies a Named Entity Recognition (NER) component. After the NER process has finished, they try to map the identified named entities to the ontology. The process is done by matching the identified entities to the labels of the concepts and instances within the ontology. Their concept allows using multiple labels for one ontological element. The problem of ambiguity is reduced by using a specific keyword property, i.e., labels which are marked as a keyword are not used for automatic annotation, but for counting instances only. Labels are only used for "instance-specific text forms". More elaborate structural mappings are not available.

Two of the domains which especially embrace ontological knowledge are the biological and medical domains since there are many large ontologies available. Many approaches exist for semantic information retrieval, e.g., [138], [139]. One of the most recent ones was developed by Koopman et al. [140]. They propose a system which uses SNOMED-CT (Systematized Nomenclature of Medicine Clinical Terms [141]) for OBIR from medical records. They argue that ontological knowledge allows a more detailed information retrieval because associational, deductive as well as abductive reasoning can be used. The concept extraction is based on MetaMap [142], which maps words from text to the corresponding elements within the unified medical language system (UMLS) metathesaurus. Next, the identified elements are mapped to their SNOMED-CT equivalents. The MetaMap process is based on a pipelined approach, in which potential candidates are identified, mapped to UMLS and disambiguated. The mapping process itself is based on a simple lexical model which is provided by UMLS. MetaMap has the ability to partially identify compound mappings (i.e., a single concept is not enough to characterize textual phrases). Representing compound mappings or more complex textual descriptions in addition to elaborate grammatical structures can be done in SE-DSNL. Further, the analysis algorithm of SE-DSNL offers the inclusion of more elaborate features like anaphora resolution.

Khelif [143] proposed a system which should help "biologists to annotate their documents". Their annotation tool "MeatAnnot" parses biological documents and tries to extract potential instances of UMLS relation. It identifies instances of such UMLS concepts which have been linked by the previously identified relationship. Again, a concrete

structural or grammatical mapping is not available. Instead, the most direct and simplistic way of associating text to semantic knowledge has been taken. The process itself is based on a standard pipeline, comprised of GATE [137], TreeTagger [144] and RASP (statistical annotation of text [145]). No information is available about which complex NLP related challenges are handled by the approach. WSD seems to be supported as part of a simplified algorithm which considers a four word window and, based on the information within this windows, tries to identify a relation within UMLS.

Another biological OBIR system is Textpresso which was developed by Müller et al. [146] [147]. Textpresso is a "textmining system for scientific literature". It consists of an ontology with a custom set of categories and subcategories, each of which has been annotated with a set of terms. Terms are allowed to be part of one category only, thereby eliminating ambiguities. The tradeoff is a very limited way of representing ontological elements. Text documents are indexed by matching the words to corresponding terms. The text database can be searched by using a combination of keywords and semantic categories. The approach is simplistic as it does not consider things like homonyms, syntactical and semantical structures etc.

Toma [148] developed a system in which documents are matched to ontologies, which are queried based on a so called "query ontology". The query ontology is created from a natural language query. The determination of the similarity between a document and an ontology is based upon the term-frequency-inverse document frequency [149]. Therefore, the number of terms, which match concepts within an ontology, have to be counted. A term matches a concept if the label of the concept matches the term. This again is a very simple approach in contrast to SE-DSNL as it can not represent synonyms and has problems with compound words or specifying syntactic structures in general. Further, the approach does not consider things like anaphora resolution or WSD.

Köhler et al. [150] also designed a system for OBIR. Their vision is to close "the gap between the HTML based internet and the RDF based vision of the semantic web". The concept relies on RDF labels for creating a bridge between text and ontological knowledge. However, they consider a certain context of both the mapped concepts (i.e., sub- and superconcepts) and the words of a document. From this information the ontological index of a document is created. Still, it is not as precise as considering the concrete semantic structure of a text.

There are more approaches, which are similar to the previously mentioned ones and are, therefore, not explained further. These are OBIR systems like AeroDAML [151] [152], Armadillo [153] [154] [155], the Knowledge and Information Management Platform KIM [156], SemTag [157] and OntoMat [158].

4.4.2. Ontology-based Information Extraction

As mentioned previously in section 2.3, Ontology based information extraction systems have similarities to SE-DSNL. In the following, we delimit several OBIE systems from SE-DSNL.

An interesting approach to OBIE has recently been proposed by Wimalasuriya [159], which is called OBCIE (Ontology-Based Components for Information Extraction). To provide an easier adaptation to different domains and ontologies, he proposes that information extraction systems should be based on independent components which only extract information belonging to specific concepts within the ontology. Depending on the circumstances, different components can be integrated in the overall framework and extract their corresponding information. This can be compared to the Construction- and Statement approach of SE-DSNL which allows the application of a specific Function in an exactly defined context, e.g., to check for new information or validate existing ones. His approach lacks a more specific definition of the mapping between semantic and linguistic information. The famous DeepQA project from IBM [160] also uses different information specific components to identify the lexical type of an answer.

Cimiano and Völker [161] presented the Text2Onto system, which is a "framework for ontology learning from textual resources". Its NLP pipeline is based upon GATE [137] and works similar to the one proposed in [136], i.e., sentence splitting and tokenization, POS tagging, lemmatizing or stemming. Next, JAPE [162] extracts the information which is necessary for the ontology learning process. This is done by applying a set of patterns, which consist of regular expressions, to identify specific information from the previous annotation process. If the expressions match, new information is created. This is similar to the Constructions of SE-DSNL, however, there are some differences: Constructions in contrast to JAPE patterns are not yet built to contain regular expressions. However, Constructions can match both semantic and syntactic information. Further, Constructions define which algorithms should be applied in which context, thereby controlling the overall parsing process. Further, Text2Onto is based on a pipeline approach, in which first different NLP components have to be applied, before information can be retrieved with JAPE. In contrast, SE-DSNL only requires the construction of a syntax tree, before in a concurrent approach different algorithms for the semantic analysis are applied, therefore making use of ontological knowledge during the analysis process and not just in the end for final validation. This partially allows to solve challenges like anaphora resolution, word sense disambiguation and information vagueness, each of which can access the results of the other algorithms at runtime.

Todirascu et al. [163] developed a system called Vulcain. The system is used for message filtering in specific domains. In contrast to many other approaches, which only apply

very simple mappings between terms and semantics, Vulcain uses a mapping system, which especially can define the relations between specific lemma and the corresponding semantic information. Hence, Vulcain uses Lexicalized Tree Adjoining Grammar (LTAG) [55]. Each elementary tree can be modified by derivation trees. This leads to strongly formalized results which can be validated against a domain specific ontology. Their approach focuses heavily on a very strong semantic representation. However, linguistic problems seem to be neglected, i.e., aspects like word sense disambiguation or anaphora detection are not being mentioned. Also, the concept does not seem to make use of ontological information during the parsing process itself. The ontological knowledge is only used to validate the final result.

Buitelaar et al. [164] developed the SOBA system (SmartWeb Ontology-based Annotation) which itself is based on the information extraction system SProUT [165]. The system is used for the "ontology-based information extraction from soccer web pages", which in turn should be used for question answering. In contrast to other OBIE systems, SOBA is capable of annotating structures besides natural language text, e.g., tabulars and image captions. For mapping text to ontological knowledge, it builds upon the LingInfo model [126], which is a predecessor of the previously introduced LexInfo model (section 3.6).

Adrian et al. [166] presented the iDocument OBIE system (which has, e.g., been used for personal knowledge acquisition [167]). iDocument provides a pipelined approach to OBIE, which is comprised of normalization, segmentation, symbol recognition (identifying which word belongs to which ontological class), instance recognition (which of the symbols is an instance of the ontology), fact recognition (facts of the text which are also part of the ontology) and finally template population (templates are used for the information extraction task, thereby trying to populate a template with the previously gathered information leads to potentially new information). As with many other OBIE systems, iDocument does not provide an exact mapping between ontological and linguistic information like LexInfo or SE-DSNL. Further, their pipeline approach does not seem to cover more elaborate linguistic problems as the ones SE-DSNL covers.

Unger and Cimiano [168] developed Pythia, a system for ontology-based question answering. They based their system on LexInfo for capturing information about "word forms, morphology, subcategorization frames and how syntactic and semantic arguments correspond to each other". Based on this information, they generate so called "grammar entries, i.e., pairs of syntactic and semantic representations". The data is transformed to a LTAG. With the information at hand they can later parse natural language questions and construct a formal representation, which is based on DUDES [169]. The parsing is done by an earley-type parsing algorithm [170] for TAG grammars. It is very much focused on natural language questions in contrast to SE-DSNL, and therefore does

not care about complex linguistic challenges like anaphora resolution or word sense disambiguation.

Li and Bontcheva [171] developed a system which in contrast to other OBIE uses machine learning concepts "by taking into account the relations between concepts" of the target ontology during the extraction process. This is used for hierarchical classification, based on Dekel [172]. Incorporating ontological knowledge in the analysis (or IE process) can be compared to the mechanisms within SE-DSNL. There are other approaches to information retrieval, which focus either on specific sources (like wikipedia [173] [174]) or use different methods for annotation (e.g., web-search [175] [176]). Other approaches have been done by Embley [105], Saggion et al. [177], Ontea by Laclavík et al. [178], Hwang [179], Yildiz and Miksch [107] and Vargas et al. [180]. All of these systems have similar approaches to OBIE as the ones presented before. We do therefore not delimit them further from SE-DSNL.

4.4.3. Semantic Annotation

Körner and Landhäußer [181] developed a system which tries to "automatically denote the implicit semantics of textual requirements". Their work is inspired by trying to create UML models from natural language text. Therefore, they annotate their texts with a standard called SAL_E [182], which defines semantic roles (see section 2.1.2.2) like AG (agents of a sentence), ACT (actions within a sentence), etc.. The automatic annotation is based on a custom NLP pipeline, which is comprised of a sentence and word splitter, POS tagger, the statistical parser for semantic annotation and NER (all componentes were taken from the Stanford NLP group [183]). In the end, the result is checked against WordNet [13] and Cyc [184] for missing information. Their approach differs from ours in such a way that it uses a standard NLP pipeline approach and does not consider more elaborate structural relations between semantics and language.

Magpie by Dzbor, Domingue and Motta [185] [186] is a concept of how web pages can be annotated semantically while browsing. It provides some simple linguistic rules for, e.g., the identification of abbreviation. The process itself relies on NER and simple string matching, no further preprocessing is done. Its successor PowerMagpie [187] improves on the problem of dynamically selecting the correct ontological source for a specific topic.

4.4.4. Conclusion

As we have shown in this section, there are no known OBIE or OBIR systems which support the same features as SE-DSNL. The distinction between existing frameworks

and SE-DSNL can be summarized as existing frameworks focusing on the retrieval and extraction of information from larger text bases without considering detailed linguistic mappings, elaborate structural analysis or the usage of ontological knowledge at runtime to validate certain information or gather new ones. Moreover, most systems do not rely on a more detailed mapping between language and semantics. Only a few systems seem to use more expressive mappings, i.e., Pythia [168] with LexInfo and SOBA [164] with LingInfo. Aside from these two no other system except for SE-DSNL yet made use of such features in a more complex domain. Regarding LexInfo, there are, however, some discussions about which scenarios the LexInfo model could be used for (e.g., McCrae et al. [188] think about linking WordNet and Wiktionary, whereas Davis et al. [189] discuss generating a lexicon from an ontology). An overview of all previously described results is available in table 4.1 and how they compare to SE-DSNL. The first row "Lexical mapping" describes whether or not an approach makes use of providing a more elaborate lexical representation of semantic information. This means that elements from the ontology should at least be representable by two or more word forms. In addition, if an approach incorporates information from a lexical database like WordNet [14], it leads to a check mark. The second row "Structural mapping" specifies if syntactic structures can be mapped to their semantic counterparts. It is, however, not sufficient if this is done by an algorithm without background knowledge. Hence, knowledge about the syntactic to semantic mapping is required. The third row "Component oriented" describes if an approach actually makes use of a component oriented concept while analyzing the available information, i.e., not one monolithic component should analyze the input sources but several components, each of which is best suited for doing a specific task in a specific context. The fourth row "WSD" specifies if an approach supports an algorithmic word sense disambiguation. The last row "Accessing sem. information" represents if an approach can also access the semantic information within an available ontology during the analysis process.

Table 4.1.: Comparison of concepts related to SE-DSNL

Approach	Lexical mapping	Structural mapping	Component oriented	WSD	Accessing sem. information
Vallet et al. [135]	✓				
Koopman et al. [140]	✓			✓	
Khelif et al. [143]	✓			✓	
Müller et al. [146] [147]	✓				
Toma [148]					
Köhler et al. [150]	✓			✓	
Kogut and Holmes [151] [152]	✓			✓	
Dingli et al. [153] [154] [155]	✓			✓	
Popov et al. [156]	✓			✓	
Dill et al. [157]	✓			✓	
Handschuh et al. [158]					
Wimalasuriya [159]			✓		
Cimiano and Völker [161]					
Todirascu et al. [163]	✓				
Buitelaar et al. [164]	✓				
Adrian et al. [166]	✓				
Unger and Cimiano [168]	✓	✓		✓	
Li and Bontcheva [171]	✓			✓	✓
Wu et al. [173] [174]	✓				
McDowell and Cafarella [175]	✓				
Cimiano et al. [176]	✓				
Embley [105]	✓				
Saggion et al. [177]	✓				
Laclavík [178]	✓				
Hwang [179]	✓				
Yildiz and Miksch [107]	✓			✓	
Vargas et al. [180]	✓			✓	✓
Körner and Landhäußer [181]	✓				
Dzbor and Domingue [185] [186]	✓				
SE-DSNL	✓	✓	✓	✓	✓

5

Semantic Spreading Activation

5.1. Introduction

Analyzing the semantic content of natural language requires solutions to different problems. A very important challenge in this context is the identification of those SemanticElements which are actually meant by the speaker or author of a written text. One of the tasks related to the challenge is known as word sense disambiguation. A more detailed introduction into the basics of Word Sense Disambiguation (WSD) and different approaches to the challenge are described in section 2.1.6.3. An example of how synonymy and homonymy can be represented with semantic knowledge, is shown in figure 5.1. The words "Car" and "Auto" represent the same SemanticElement 'Car' and are therefore synonyms. The words "Auto" as well as "X1" are homonyms because they have multiple meanings ("Auto" represents the SemanticElements 'X1' and 'Car', whereas the meaning of "X1" can be either the SemanticElement 'X1' or 'X1 TDI'). All known approaches to

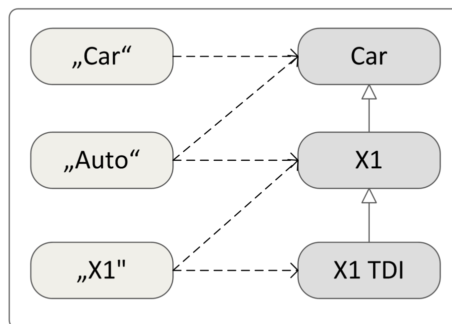


Figure 5.1.: Homonym and Synonym Relation

WSD like machine learning or graph-based methods have in common that they select the most probable sense for a word from a set of existing SemanticElements. For the example in figure 5.1 this would mean that a user trying to disambiguate the word "Auto" would receive a set containing both SemanticElements 'Car' and 'X1' and is only allowed to choose from these two SemanticElements. However, there are cases where this is not enough. Humans tend to be ambiguous regarding not only the use of synonyms and homonyms, but also of hyponymy and hypernymy (this was more broadly introduced as 'Vagueness' in section 2.1.4). Hyponymy means that the semantic content of a word is a specialization of the semantic content of another word (hypernymy means the opposite). In figure 5.1 a hyponymy relation exists between the words "X1" and "Auto", because "X1" represents the SemanticElement 'X1 TDI' and "Auto" represents the SemanticElement 'Car', which is a parent of 'X1 TDI'.

A challenge is that an author might use the word "Car", but actually mean the SemanticElement 'X1 TDI'. It can be seen that there is no direct relation between the word "Car" and the SemanticElement 'X1 TDI' in figure 5.1. A reader, however, can most probably

infer the correct SemanticElements based on the context. Classical word sense disambiguation algorithms are not capable of solving the problem, as they either do not have the necessary knowledge (i.e., a clear description of both lexical / syntactic information on one side and semantic information on the other side) or are not capable of using the knowledge accordingly. Another problem in which a word has a different meaning than the ones which it is normally being used for, is the phenomenon of reference transfer (as described in section 2.1.4). The problem is even harder to solve than vagueness.

The algorithm that we developed and describe in the following sections does not directly disambiguate a complete sentence. Instead its primary task is to evaluate the semantic relatedness of a given input, consisting of either a tuple or a triple of SemanticElements (those input sets can be enriched with additional context SemanticElements). As part of the semantic relation evaluation, additional SemanticElements might be found which could be more appropriate than the initial input (because of either reference transfer or vagueness). Those SemanticElements represent alternatives to the original input of the algorithm. In the following section we describe an algorithm which is capable of solving those problems and more. Therefore, first our requirements are stated before the algorithm itself is introduced.

5.2. Requirements

The section describes in detail the requirements which the spreading activation algorithm needs to fulfill.

1. Analyzing text requires disambiguating the senses of the single words. Hence, it is necessary to have a metric which indicates if different SemanticElements are semantically related to each other. We assume that the information helps us in solving the WSD problem. The algorithm should return a value between 0 and 1 which represents whether a specific information is available within the ontology and how closely it is related. 1 indicates that there definitely is such a relation available. 0 means that no information could be found.

2. As humans tend to overgeneralize, the approach should be capable of making a set of SemanticElements as specific as possible, i.e., if a person talks about the SemanticElement 'Car' in figure 5.2, but further mentions specific attributes (e.g., the color 'Red'), it is clear to his communication partner, which type of car is meant (i.e., the 'E3'). The process should be mimicked by the algorithm.

3. Humans sometimes only mention specific attributes of what they actually refer to, i.e., in contrast to the previous requirement they don't mention a word which directly represents the SemanticElement 'Car' but may refer to the SemanticElement by one of its attributes. An example could be 'I drive a red one'. The reader infers that the author most probably meant a car (or in this case the more specific element 'E3'). Our approach should try to identify and solve the problem as well.

5.3. Example

In order to better understand the algorithm it is illustrated by a small example using figure 5.2. The ontology describes a small excerpt from a car domain. It consists of SemanticElements like 'Person', 'Car', 'Color' and different specializations of those SemanticElements. For the example we define $A_t := \{\text{'Driver'}, \text{'Drive'}, \text{'Red'}\}$ and $A_c := \{\}$, i.e., a full triple and no context elements. The initial state of O_{se} can be seen in figure 5.3 in the upper left part. The green dots represent all elements which correspond to the input A.

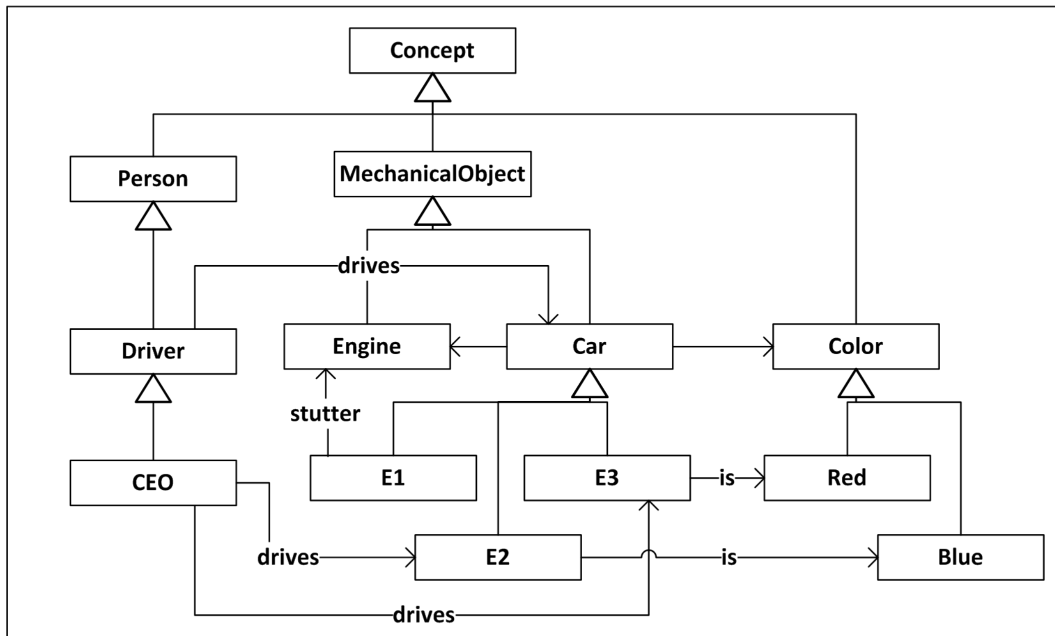


Figure 5.2.: Ontology example

5.4. Definitions

The definition is based upon the definitions 1 (Ontology) and 2 (SemanticScope O_{se}). The algorithm requires a specific as specified in the following:

Definition 29 (*Input*)

Let

$$A := (A_t, A_c) \tag{5.1a}$$

$$A_t := \left(e_s, e_y, e_t\right) \tag{5.1b}$$

be the input to the spreading activation process. A_t represents a triple, where $e_s, e_y, e_t \in O_{se}$ and $A_c \subseteq O_{se}$. e_s is the source SemanticElement, e_y is the type of an Association which has e_s as its source and e_t specifies the target of the Association. A_c is a set of SemanticElements which act as additional information (context) to the spreading process.

An example for A_t is an element 'Driver' as e_s, 'Drive' as e_y and 'Vehicle' as e_t. It should be noted that A_t can also consist of a tuple $\{e_s, e_y\}$ or $\{e_s, e_t\}$ only. A_c is always optional.

The algorithm does not create a new spreading activation network (SAN) but instead relies on using the available graph structure within O_{se}. However, in order to save all the information which are being created by the spreading process a container element is needed.

Definition 30 (*Token Container*)

Let

$$k := \{k.e, k.T, k.act, k.pt, k.d\} \tag{5.2}$$

be a token container. The entities of the set are specified as follows:

1. $k.e \in O_{se}$ is a SemanticElement which represents the ID of the token container (the ID is unique because one concept may only be represented by one SemanticElement within O_{se}).
2. $k.T$ is defined as a set $k.T := \{t_1..t_n\}$. k represents a container for all the tokens which have reached the SemanticElement $k.e$
3. $k.act$ indicates if the SemanticElement $k.e \in A_t$
4. $k.pt$ is an attribute which counts the number of times that the container has been part of a backpropagation process.

5. $k.d$ represents the depth of the SemanticElement $k.e$ within the Generalization hierarchy of O_{se}. The depth value is calculated as the position of $k.e$ relative to the length of the longest branch it is located in.

In further references if k is mentioned as being part of A_t or another set of SemanticElements, this actually means $k.e$, which should be contained in the corresponding SemanticElement set. In the following, we refer to k_s as the container of e_s, k_t as the container of e_t and k_y as the container of e_y.

The tokens being passed around in O_{se} are defined as follows:

Definition 31 (*Token*)

Let

$$t := \{t.orig, t.start, t.pos, t.en, t.s, t.dir, t.pred\} \tag{5.3}$$

be a token. The different entities of the set contain the following information:

1. $t.orig$ holds a reference to its original container which must be a container of one of the SemanticElements in A.
2. $t.start$ is a reference to the container where the token originally started from (this can, but does not have to be the original container; it may be a container whose SemanticElement is related to the SemanticElement of $t.orig$ via Generalization)
3. $t.pos$ is the container representing the current position of the token.
4. $t.en$ indicates the remaining energy of the token if the energy drops below a certain threshold the token can not spread any further.
5. $t.s$ describes the steps the token has already moved within O_{se}. The value increases on every new element except for Generalizations.
6. $t.dir$ defines the direction in which a token moves. Values can be up or down (within the generalization hierarchy), sidewards (i.e., on an association) as well as unknown (in case of the token being placed on the very first container).
7. $t.pred$ is the predecessor token of the token or null in case of this being an initial token.

A token moves over the structure in O_{se} and creates a so called path.

Definition 32 (*Token Path*)

Let t_m be a token. Then

$$path(t_m) := [t_m, t_{m-1}, .., t_1] \tag{5.4}$$

returns a list of tokens where $t_{m-1} = t_m.pred$, $t_{m-2} = t_m.pred.pred$ etc. and $t_1.pred = null$. A path contains a SemanticElement $e \in O_{se}$, if for a path $p := path(t_m)$ the following predicate holds:

$$containsElem(p, e) \Leftrightarrow \exists t \in p\, equal(t.pos.e, e) \tag{5.5}$$

where *equal* checks if two elements are identical. This means that one token within a path p is located on a token container whose SemanticElement $t.pos.e = e$.

Finally the algorithm returns an output. The structure is defined as follows:

Definition 33 (*Algorithm Result Structure*)

Let

$$R := \{R.v, R.p_s, R.p_t\} \tag{5.6}$$

be the set representing the output of the spreading activation process.
$R.v$ is a value within $[0, 1]$ representing the semantic relatedness of SemanticElements within A_t with respect to O_{se}.
$R.p_s \in O_{se}$ is a SemanticElement which represents a proposal of the algorithm as to what the user could have actually meant by referencing e_s based on the information in O_{se}. Note that $R.p_s$ can either be specification of e_s (i.e., $R.p_s \subset e_s$) or a completely new element, i.e., $R.p_s \not\subset e_s$.
$R.p_t \in O_{se}$ is analogue to $R.p_s$ except for that it is a proposal for e_t.

The structure of the result already supports the previous three requirements. First, $R.v$ contains the value for requirement 1, whereas $R.p_s$ and $R.p_t$ can contain proposals as required by requirements 2 and 3.

5.5. Initialize tokens

The algorithm is initialized based on input A. It first creates a token container for each $e \in A$ as seen in algorithm 11 (e.g., $INIT(e_s, 1.5)$). The initialization is based on the Generalization hierarchy of the SemanticElement e. All elements within A are basically treated the same (i.e., their initial energy value is the same). The only exception is e_s which receives a higher initial energy value than the remaining elements, since we want to know whether there is a path from the source to the target SemanticElement. Therefore, the higher energy value allows them to move farther.

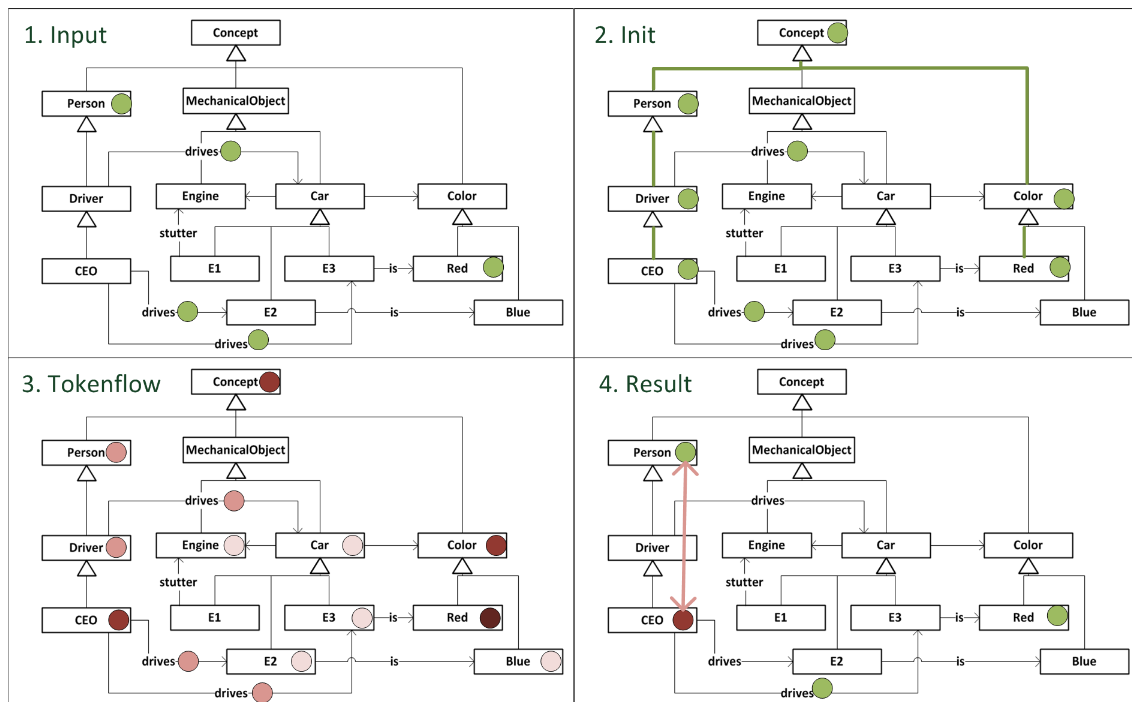


Figure 5.3.: The four phases of the first spreading activation iteration

As can be seen in algorithm 11 the initialization is done in both Generalization directions (*INITGENUP* means that the initialization is done to the top of the Generalization hierarchy, i.e., more general SemanticElements are initialized; In contrast *INITGENDOWN* initializes more specific SemanticElements). The reason is that humans tend to be ambiguous while communicating and often use more generalized terms than they actually mean (see requirement 2 in section 5.2). Only the context of a word helps in deciding which SemanticElement they actually refer to. Hence, the initialization down the hierarchy helps to initiate all elements, which eventually are meant by a human, whereas the call upwards initializes all those elements which may contain the corresponding semantic information that a SemanticElement e inherited. The information is necessary in order to correctly analyze the current input.

INITGENUP initializes a single SemanticElement by creating a token container for every SemanticElement in the upwards Generalization hierarchy. It further creates the initial tokens for each of these SemanticElements (*INITGENDOWN* works analogously). It must be noted that every SemanticElement which is initialized by *INITGENUP* is treated as being part of the original input. Therefore, the *k.act* attribute of their containers is set to true since each of these SemanticElements could be the carrier of the information we search for.

Figure 5.3 shows in the upper right what O_{se} looks like after the initialization. Each SemanticElement of input A has been initialized which also activates all elements of the corresponding hierarchy branches (indicated by the green colored circle).

Algorithm 11 Initialization

Input: e: A SemanticElement of the input A
ENERGY: The initial energy value for all tokens
Effect: Creates tokens and token containers for all children and parents of the SemanticElement e

```
1: procedure INIT(e, ENERGY)
2:     INITGENUP(e, e, ENERGY, null)
3:     INITGENDOWN(e, e, ENERGY, null)
4: end procedure
```

5.6. Create the Token flow

The set of initial token containers and tokens has been created. Now the token flow itself has to be calculated. The process which is discretized in single phases is shown in algorithm 12. Each current token generation $T_{current}$ leads to a new token generation T_{next} which is processed after every token from the current generation has been analyzed. This process is required as the *POSTPROCESS* call initializes a back propagation mechanism. A non discretized process would yield indeterministic results.

Algorithm 12 Process Tokens

Input:
Effect: Iterates over the current token set $T_{current}$ and creates new tokens until no more tokens are available in T_{next}

```
 1: procedure PROCESSTOKENS
 2:     while T_next.size ≠ 0 do
 3:         T_current := T_current ∪ T_next
 4:         T_next := {}
 5:         PREPROCESS
 6:         CREATETOKENFLOW(T_current, T_next)
 7:         POSTPROCESS
 8:         T_current := {}
 9:     end while
10: end procedure
```

CREATETOKENFLOW is called with the set of current as well as next tokens. For every token $t \in T_{current}$ it first checks the following three conditions:

1. If *t.dir* is unknown, the token is allowed to move both on Associations (sidewards) as well as on Generalizations (up / down) in O_{se}. A token is, however, not allowed to move upwards, if its previous direction was down or sidewards before. The cause for these restrictions is that the tokens otherwise could reach irrelevant or false SemanticElements. All those direction changes of course depend on the Associations and Generalizations available on *t.pos*.

2. *t.s* is not allowed to exceed a certain threshold. If it does it can not move any farther. We assume that SemanticElements which represent not-related real world objects, may also have a long or no path at all between them within O_{se}. The length of the current token path is measured by the number of SemanticElements which a token has already visited.

3. Finally *t.en* indicates the remaining energy of a token. If *t.en* drops below a certain threshold the token can not continue. However, *t.en* can be increased during the back propagation mechanism (which is described later).

Only if all three conditions are evaluated successfully new tokens are generated (i.e., tokens for the next relation which t moves on, as well as the target SemanticElement of the relation) and added to the T_{next} set. The energy of a new token t_{new} is based on the current tokens $t.en$ attribute and is decreased by a constant value. However, if the container of a relation has been activated (i.e., $k.act = true$), no energy is subtracted from the energy of the new token. The process allows us to enhance the energy of paths which are likely to be more relevant to the spreading activation input. Finally, the predecessor attribute $t_{new}.pred$ is set to the token t.
Next the *POSTPROCESS* method is called. It starts the back propagation mechanism on all containers whose *k.act* attribute is set to *true* and which have received new tokens in the last token flow phase. Each token on such a container gains an increase of its energy value:

$$t.en := t.en + (EN_{MAX} - t.en) * T_e \tag{5.7}$$

where EN_{MAX} denotes the maximum energy a token can have and T_e is a constant factor between 0 and 1 which is used to weight the value. The mechanism is recursively executed on the predecessors of the token. By activating the propagation mechanism on such containers which have been part of A, only those token paths are strengthened which have a relation to the input A. We therefore assume that these paths are also more likely related to the final result. Further, even if some of the elements within A_t may not be ideal (e.g., they might be underspecified), we assume that the intended SemanticElements may still be connected to the SemanticElements within the activated paths.

Figure 5.3 shows the result of the token flow process in the bottom left. The colors indicate the weight of the corresponding concept container (dark red means a heavy weight whereas light red represents a lite weight). It can be seen that the SemanticElement 'CEO' (i.e., a more specific element of the e_s element 'Person') has received a heavy weight. This is due to tokens from 'CEO' which arrived at the concept 'Red'. Their energy has been increased during the backpropagation mechanism. In general, the elements 'CEO' → 'drives' → 'E3' → 'is' → 'Red' received a high activation value.

5.7. Analyze the token flow

The final step involves gathering the results from the token flow process. The algorithm starts by identifying more specific SemanticElements of the actual input. This step is required to reduce vagueness (see requirement 2). The analysis process creates a list K_s in which all token containers are collected whose SemanticElement $k.e$ is more specific than a SemanticElement $e \in A_t$, i.e., $k.e \subset e$. Next the most relevant tokens of each token container $k \in K_s$ are identified. Relevant tokens are those which were sent from another SemanticElement $e \in A_t$, where $e \neq k.e$. The algorithm sorts K_s based on

1. the number of relevant tokens which reached this container (i.e., tokens from SemanticElements of A_t/e),
2. the energy value of the relevant tokens (higher values imply a bigger relevance to the input A),
3. activation times (i.e., how often the container was activated in the *POSTPROCESS* method; the more often a container was activated the more relevant it is to the input A)
4. and the depth of the SemanticElements within its Generalization branch (more specificity is better).

The process picks the best token container $k \in K_s$ for each $e \in A_t$ (or none, if there is no token container available for an element $e \in A_t$). The SemanticElement $k.e$ represents a proposal for a more specific SemanticElement of the input element e. Thus, if there is a token container k_s which is more specific than e_s, the value $k_s.e$ is assigned to $R.p_s$ (the same happens for $R.p_t$ if there is a more specific token container k_t available). However, if there are multiple token containers within K_s for an element $e \in A_t$ all of which received the same highest rating value, no container is picked. The reason is that this would contradict the idea of reducing vagueness and ambiguity.

All information necessary for the final result has been calculated. However, it might be the case that the result is not perfect, i.e., the initial input represents a case of reference transfer (i.e., requirement 3). For such a situation a heuristic is used which identifies those cases and tries to find a better solution. Still, some restrictions have to be made: An 'imperfect' situation can only be identified if e_s, e_y and e_t are available in A_t. If there would only be two SemanticElements available in A_t, there are too few information, which would lead the heuristic to imprecise decisions. Further only situations in which either e_s or e_t are wrong can be detected. For the following we refer to the example from section 5.2 in which case e_t is 'wrong' (as it references 'Red' instead of 'Car').

First, the algorithm collects all available Associations $rinO_{se}$ which are of type e_y and reference either e_t or e_s. Those are stored in the list L_p. L_p is sorted based on the number of times that the token containers, which represent the source and target SemanticElements of the Associations, have been activated (i.e., the value of the *k.pt* attribute). The assumption is that the more often an Association has been activated, the more relevant it is to the input A_t. Next the algorithm tries to find an Association $r \in L_p$ which has a source and a target SemanticElement that matches e_s and e_t:

$$(r.s \subseteq e_s \vee r.s \supseteq e_s) \wedge (r.t \subseteq e_t \vee r.t \supseteq e_t) \tag{5.8}$$

where $r.s$ is the source SemanticElement and $r.t$ is the target SemanticElement of the Association, i.e., the SemanticElements are verified for either being a child or a parent element. The reason is that humans are ambiguous with how they express themselves, i.e., they are sometimes very specific and, in other cases, very vague. Hence, the heuristic needs to be tolerant towards both these types of human expressions. If there is an Association that matches condition 5.8 we assume that the algorithm seemingly has been used on a correct input and the spreading activation terminates. If, however, no Association in A_p matches the condition, the algorithm reinitializes. The behavior is based on the assumption that if no Association can be found in O_{se} which represents the complete input triple e_s, e_y and e_t this might be a case of reference transfer which must be validated. Therefore, the best Association of L_p (i.e., the one with the highest rated source and target SemanticElements) is used because based on the current token flow the Association represents the closest match to the input A_t. Next, the spreading activation is reinitialized with a new A':

1. The 'wrong' SemanticElement within A_t (either e_s or e_t) is replaced with a SemanticElement ($r.s$ or $r.t$) of the best Association $r \in L_p$ (in our example the element e_t 'Red' is replaced with $r.t$ 'E3').

2. The old element which has been replaced (either e_s or e_t) is added to the list of context elements in A'_c. It might provide helpful information for the next spreading activation iteration. The assumption is that the user had a reason to mention the specific SemanticElement in the natural language input text. Hence, the information should not be thrown away as tokens from this SemanticElement are probably closely related to the actually intended SemanticElement and may therefore provide the final clue as to which SemanticElement the user was referring to.

In our example, the process restarts with the new input $A'_t := \{$'CEO', 'Drives', 'E3'$\}$ and $A'_c := \{$'Red'$\}$. Note that A'_t seemingly contains two new SemanticElements: 'CEO' as a proposal for 'Person' and 'E3' for 'Red'. However, 'CEO' is a specialization of 'Person' and therefore not a completely new concept (as specified in definition 33). In the second

iteration the same steps are executed. If a seemingly correct result can be found (for more details, see section 5.9), the algorithm returns the new result. If, however, the conditions for starting the heuristic would match again, the process stops. The reasons for only allowing two iterations is that we could not detect any improvements if the heuristic would be allowed to trigger a third or fourth iteration. On the contrary, sometimes this lead to completely irrelevant false positives. Therefore, the algorithm returns the best result from the first two iterations only.

The results of the second iteration for the previous example can be seen in figure 5.4. Again, first the corresponding SemanticElements and Associations are marked as input elements. Next the initialization takes place. The token flow this time leads to the SemanticElements 'Driver', 'Car', 'CEO' and 'E3'. The analysis process finally can not identify any more specific SemanticElements than those which have been part of A'. Further as A' has been found within O_{se} (meaning that there is a triple 'CEO' → 'drives' → 'E3') the algorithm terminates. The variable $R.p_s$ is set to the SemanticElement 'CEO'. Also, the SemanticElement 'E3' is assigned to $R.p_t$

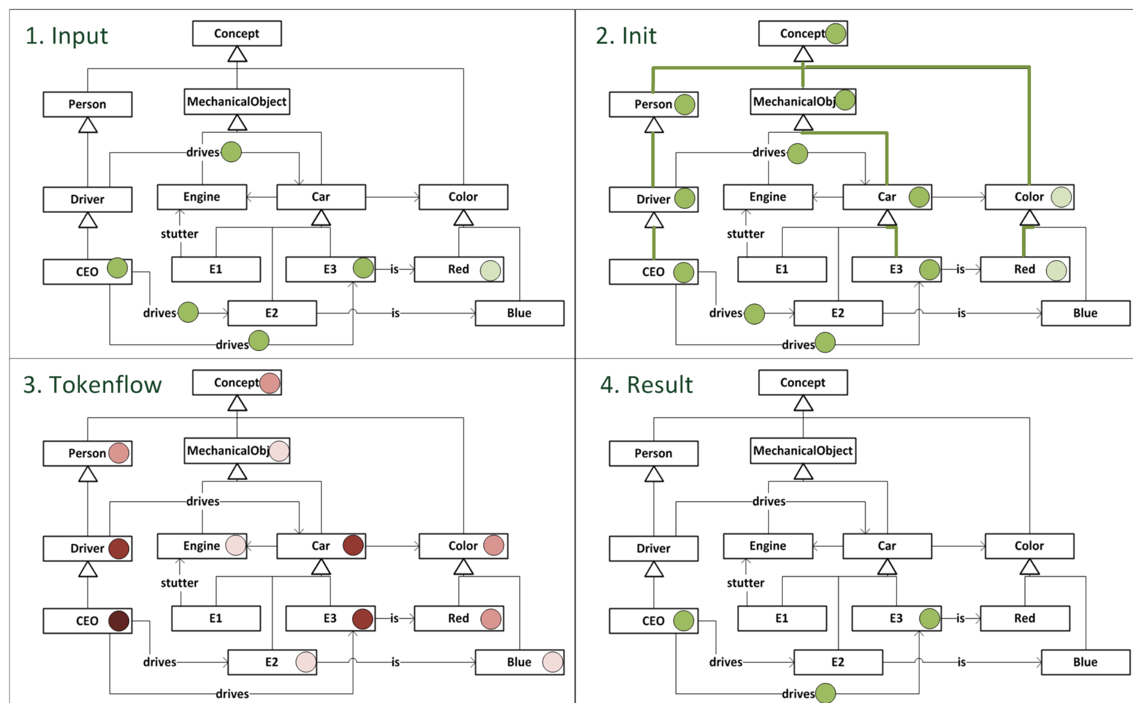


Figure 5.4.: The four phases of the second spreading activation iteration

The last algorithm step calculates the value $R.v$ which represents if and how well the information of A_t exists within O_{se}. Two different cases have to be distinguished:

1. The first case occurs, if the heuristic did not step in, i.e., the initial source and target elements are still the same. In this case, the process searches for a token t from the

target container k_t which equals $t.start = e_s$, i.e., it has the source container as its starting position. If such a token can be found the computation of $R.v$ depends on the average energy of the token compared to $t.s$ (the number of steps a token has moved).

2. If the original source or target SemanticElements have been replaced by a new SemanticElement (due to the heuristic and a second iteration), a different calculation is required. In this case, the value depends on the semantic similarity (based on the minimum distance) between the initial elements e_s or e_t from A_t and the replacement SemanticElements in A'_t.

5.8. Termination

We are going to have a look at the most sensitive parts of the algorithm concerning its termination. The explanation is separated into three sections according to the different parts of the algorithm. We assume that the spreading activation algorithm is always only executed on a valid SE-DSNL model (definition 14).

Initialization The initialization phase consists of creating token containers as well as the initial tokens for each SemanticElement $e \in A$. The process does not terminate if there is a circle within the Generalization hierarchy of O_{se}. However, guideline 3 in section 3.5.4 defined that Generalization hierarchies are not allowed to form circles within a valid SE-DSNL model. Thus, the initialization phase terminates.

Token Flow The next step is the token flow process. T_{next} must be empty in order for the loop in algorithm 12 to terminate. This is the case if *createTokenFlow* can not create any new tokens based on those which are available in $T_{current}$. There are two different cases that have to be analyzed:

1. If $T_{current} \equiv \emptyset$, then $T_{next} \equiv \emptyset$. This case is trivial as tokens are only being created based on existing tokens. Hence, if $T_{current}$ is empty, T_{next} also remains empty.
2. The token flow stops at a given time, therefore T_{next} is not filled with any new tokens. Three conditions have been stated in section 5.6, which can restrict tokens from moving. Hence, if a token can not continue then T_{next} will not be filled with new tokens.

If either of both cases is met at runtime, the algorithm will terminate. In the following, we will elaborate the second case, which is built upon the three conditions from section 5.6. The condition which matches every token, is the restriction of the maximum number of steps $t.s$ a token is allowed to move. We will in the following show that every token must in a worst case scenario reach the maximum number of allowed steps. The attribute is based on the $t.s$ value of the tokens predecessor and is always larger in a new token than its predecessor, i.e., $t.s > t.pred.s$, because $t.steps := t.pred.s + 1$. Therefore, if the threshold for not creating new tokens is set to a value n, the algorithm will not create any new tokens after a maximum of $n + 1$ algorithm iterations. Hence, the algorithm will terminate because T_{next} will remain empty.

Result Analysis The final phase of the algorithm is the analysis step. It consists of three different parts. The first one is the identification of more specific elements. The consid-

erations are similar to the initialization phase. As the identification phase is based on analyzing the downwards faced generalization hierarchy of a Semanticelement it has to terminate because the Generalization hierarchy does not contain circles and is therefore finite.

The next part is the heuristic itself. The core part of it is the collection of specific Associations from O_{se} and their evaluation. This is a simple process which analyzes a finite set of elements and must, therefore, also terminate. Further, the heuristic is only applied once, therefore the overall algorithm has a maximum of two iterations.

The final part is the computation of a value indicating if A_t could be found in O. Two different cases have to be distinguished. The first one steps in if the heuristic was not applied. There, only the tokens of token container k_t are analyzed (which again is a finite set). In the second case, the heuristic is used. The minimum distance between two element $e_1 \in O_{se}$ and $e_2 \in O_{se}$ is computed. It is based on Dijkstra which has been proven to terminate. Further the graph is based on O_{se} which represents a finite set of elements.

5.9. Result Computation

In this section we refine how and why the spreading activation algorithm calculates valid results, i.e., results which comply to specific conditions. Therefore, we define conditions and analyze how the results match these conditions.
The structure of the algorithms result R was specified in definition 33. $R.p_s$ and $R.p_t$ are treated equally in the following, therefore we just write about $R.p$ if we refer to $R.p_s$ or $R.p_t$. We show for $R.p$ and $R.v$ what a correct result is.

Computation of R.v The value $R.v$ must lie within $[0,1]$ (i.e., $0 \le R.v \le 1$) as we defined in the requirements. It represents if and how well the information in A_t is available O_{se}. 0 indicates that the information could not be found (note that we don't say that it does not exist, we simply state that it does not exist in O_{se}). 1 indicates the definite existence of A_t in O_{se}. Values which lie within $]0,1[$ (i.e., $0 < R.v < 1$) are fuzzy values and state that a certain similarity between the elements in A_t and O_{se} could be found (the higher the value the higher the similarity). We show in the following that $R.v$ always is within $[0,1]$ and that a smaller value represents a lower similarity between A and O.

The computation of the value is based on two different cases:

1. In the first case the heuristic has not been applied. therefore the value $R.v$ depends on a token t and its path $p := path(t)$. Ideally the number of steps $t.s$ is 2 and the following holds for t:

   $$containsElem(p, e_s) \wedge containsElem(p, e_t) \wedge containsElem(p, e_y)$$

   i.e., the token t moved over all triple elements within A_t. Further, 2 represents a smallest possible path, i.e., the token moved over all elements within A_t with the smallest number of steps. This case leads to an $R.v$ value of 1.

2. In the second case $t.s > 2$ and t still moved over all elements in A_t on its path, hence the computation depends on the path energy $t.en$. The value is normalized and multiplied with a constant factor between $(0,1)$. This immediately leads to $R.v$ also being between $(0,1)$ because both the normalized value and the constant factor have a value in $(0,1)$. The reason for the approach is that tokens which may have a high average path energy may result in a value of 1. However, if we know that the path of a token to the target element e_t was longer than the optimal minimum value 2, A_t is not directly represented within O. Hence, $R.v$ is further lowered with a constant factor. The last case is that e_y could not be found on the path of t (i.e., $\neg containsElem(p, e_y)$) or that no token moved from k_s to k_t at all. In this case $R.v$ is set to 0, as no semantic connection could be found between A_t and O_{se}.

In case that the heuristic is applied $R.v$ depends on the similarity between e_s and e'_s or e_t and e'_t respectively. We only describe the correctness for the case e_s to e'_s (the other one is analogue). In a heuristic induced iteration the input A of the first iteration consists of elements A_t of which two are known to be related directly (i.e., either e_s and e_y or e_y and e_t). In the input A' for the second iteration all three elements in A'_t are connected directly. The value for $R.v$ is now computed based on the semantic distance between $R.p'_s$ and e_s, i.e., the source element which has been proposed in the second iteration and the original input source SemanticElement e_s of the very first iteration. The distance is normalized based on a maximum path length which indicates a minimum of semantic relatedness between two SemanticElements (the longer the distance between two SemanticElements is the lower the semantic relation value $R.v$ between both of them becomes):

$$R.v := 1 - \frac{dist(e_s, R.p'_s)}{d_{MAX}} \tag{5.9}$$

where *dist* measures the distance between two SemanticElements in O_{se}, and d_{MAX} is a constant representing the maximum allowed distance within which two SemanticElements are considered to be semantically related. As the distance between e_s and $R.p'_s$ is always greater or equal than 1 (if the distance would be 0 the heuristic would not have been applied in the first case and it would not have come to this situation), $R.v$ can never be 1. Instead $R.v$ is in $[0, 1)$. This result is correct because the heuristic only 'guesses' a probably better suited result, therefore a value of 1 is not allowed in case that the heuristic has been applied. $R.v$ can only be 1 if A_t was found in O_{se}.

Computation of R.p The algorithm sometimes identifies SemanticElements which seem to be more likely than either e_s or e_t (as described previously in section 5.7). If such elements exist, they are returned as $R.p_s$ and $R.p_t$. $R.p_s$ and $R.p_t$ are only 'guesses' and therefore showing their validity is difficult. We first introduce an assumption which we base the validity on: The token container of a proposed SemanticElement $R.p$ has a larger *k.pt* value than the token containers of one of its Generalization hierarchy siblings or parents. We explain the reason for this assumption in the following.

Let $R.p_s$ be a SemanticElement which is more specific than e_s, i.e., $R.p_s \subset e_s$. It therefore contains at least the same information as e_s. Each of the elements within A as well as their children (and therefore also $R.p_s$) has been initialized and used for the spreading process. During the process the back propagation may have been initiated several times, which increased the *k.pt* attribute of different token containers (as explained in section 5.6). Tokens can only move a certain distance (because of a limited number of steps and

energy) as well as in a certain direction. Therefore, the effects of the backpropagation mechanism are restricted to a certain area. We assume that SemanticElements within O_{se} are closely related to each other for a reason, i.e., their real world counterparts are related in the same way. Hence, $k_m.pt > 0$ indicates a semantic relation between the token container $e_m \in A_t$ and other concepts $e \in A \setminus e_m$. The same is true for an element $R.p$, i.e., if it is closely connected to the elements in A, the value $k_p.pt$ will be larger than 0. Figure 5.5 shows a small example. There, A_t consists of the elements 'Engine' and 'Car' only. In the initialization phase the SemanticElements 'E1' and 'E2' are activated. However, $k_{E1}.pt > k_{E2}.pt$ because k_{E1} is activated once (a token from 'E1' reaches the SemanticElement 'Engine' which was part of A). Tokens from 'E3' can not reach either 'Engine' or 'Car', therefore $k_{E2}.pt = 0$ and 'E1' is the proposed element $R.p$ for the input element 'Car'.

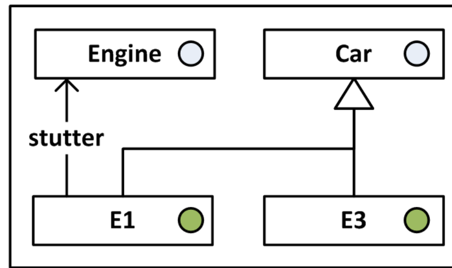


Figure 5.5.: Example for the computation of R.p

In the following, we define several conditions which have to be fulfilled by the concepts $R.p_s$ and $R.p_t$. The conditions for $R.p_t$ work analogously.

1. $R.p_s \subset e_s \vee R.p_s \subseteq e'_s$, i.e., $R.p_s$ must be a specialization of either e_s (the source SemanticElement of A, i.e., the input of the first iteration) or more specific or equal than e'_s (the source SemanticElement of A', i.e., the input of the second iteration). The reason why $R.p_s$ can be equal to e'_s is that e'_s itself must be more specific than e_s, i.e., it is a proposal of the heuristic.

2. In order for an element $R.p_s$ to be chosen as a proposal for e_s it has to correspond to the following condition:

$$k_{ps}.pt > 0 \wedge k_{ps}.pt > k_{es}.pt \wedge \forall k_m \in children(k_{es}.e)\; k_m \neq k_{ps} \wedge k_{ps}.pt > k_m.pt \quad (5.10)$$

   where k_{ps} is the container of $R.p_s$, k_{es} is the container of e_s and *children* returns the token containers of all children of the SemanticElement $k_{es}.e$. The condition basically specifies that an element $R.p_s$ is a valid proposal for e_s if it has been activated more often than any other child of e_s.

$R.p_s$ is valid if it satisfies all of the those conditions. The first condition requires $R.p_s$ to

be more specific than e_s or e'_s. As the algorithm for identifying $R.p_s$ only analyzes the children of e_s, the condition is easily met.
The second condition requires the token container of $R.p_s$ to have been part of more propagations than the container of e_s or any child of e_s. This is easily proven by the mechanism of the algorithm. It collects all children of the container k_{es} in a list and sorts it based on the containers *pt* attributes. If several children have the same highest value, $R.p_s$ is not set. If there is exactly one child k_i whose $k_i.pt$ attribute is also higher than that of $k_{es}.pt$, $k_i.e$ is assigned to $R.p_s$.

Application of the Heuristic In the previous section we only looked at the case in which $R.p_s \subseteq e_s$. In this section we want to show that the results of the heuristic are valid. We assume that if for an input A the result value $R.v$ is in $(0,1)$ (i.e., all elements of A could be found in O_{se} but its not a direct triple), there might be a better suited element e'_s for e_s (or e'_t for e_t). This could lead to $R.v = 1$ if the element would, e.g., be part of an input $A := \{e'_s, e_y, e_t\}$ for a first iteration of the algorithm. This means that by replacing either e_s or e_t in A a full triple should be found within O_{se} which might better represent what the user actually meant. The algorithm has to fulfill the following conditions (note that the case in which e_t is being replaced works analogously):

1. $rel(e_y, e_t) \vee rel(e_s, e_y)$, where *rel* checks if two or more elements are directly related to each other, i.e., the distance between all of them must be 1.

2. $rel(e'_s, e_y, e_t)$, i.e., the replacement element e'_s must be directly related to an Association type element e_y which is directly related to the SemanticElement e_t.

3. An element which is proposed as part of a second iteration must fulfill the following condition:

$$\forall e \in O_{se}\ e \neq e'_s \wedge rel(e, e_y, e_t) \wedge weight(e'_s, e_y, e_t) > weight(e, e_y, e_t) \tag{5.11}$$

   where

$$weight(e'_s, e_y, e_t) := k_s.pt + k_y.pt + k_t.pt \tag{5.12}$$

   where k_s is the token container which represents e'_s, k_y represents e_y and k_t represents e_t. Condition 5.11 specifies that there may not be any other Association of type e_y from an arbitrary SemanticElement e to the SemanticElement e_t which has a higher weight than the Association starting from element e'_s.

In the following, we show that the results of a second algorithm iteration always fulfill the previously specified conditions. We therefore look at the specification of the algorithm, especially the specification of the heuristic in section 5.7. As it has been stated previously,

the heuristic identifies only such cases in which either e_s or e_t are incorrect. This is done by identifying a list of Associations L_p of type e_y which are connected to either the SemanticElement e_s or e_t (thereby fulfilling condition 1). The algorithm sorts L_p based on the *pt* attributes of the token containers which represent the source and target elements of $r \in L_p$. The element e_s' which replaces e_s must be part of one of the Associations $r \in L_p$, therefore condition 2 is fulfilled. Further e_s' is the source element of Association $r_{best} \in L_p$ with the highest rank. This fulfills condition 3 because L_p is sorted based on the *pt* value of its token containers. Therefore, r_{best} is the best ranked Association if the *pt* value of its elements is larger than those of the remaining relations in L_p. r_{best} further represents a triple which is obviously a part of O_{se} (because it has been taken from O_{se}). If the algorithm is therefore initiated with the triple which represents r_{best}, it returns 1 (according to section 5.9). Hence, the result of the heuristic iteration is correct.

5.10. Related Work

The origins of spreading activation based approaches date back to 1962, when Quillian [190] developed a theory for semantic memory search. In 1968, Quillian introduced the term "Semantic Network" [191]. It became the most used structure for associative information retrieval processes. In 1975, Collins and Loftus [192] extended the work by Quillian such that it could be used for experiments about semantic memory. Their work is the basis for all of todays spreading activation approaches. Schiel [193] introduced the concept of abstraction into semantic networks. Ontologies posses similar features, making them alike to semantic networks. Therefore, we present approaches which use spreading activation both on semantic networks or (more recently) on ontologies. The focus is laid upon spreading activation within word sense disambiguation scenarios, which is a common use for spreading activation within natural language related tasks [194].

The most closely related concept to our approach seems to be that of Kleb and Abecker ([195]), which disambiguate word senses based on RDF graphs. They state homonymy and synonymy as their main problems (whereas we differentiate some more problems as stated previously). Their approach does, however, not consider the problems of vagueness or reference transfer.

Tsatsaronis et al. ([196], [197]) describe a spreading activation based approach, which uses the information from a thesauri to create a spreading activation network (SAN) for WSD. Their concept is used to disambiguate complete sentences at once. The background knowledge used is from WordNet 2. In ([198]) Tstsaronis et al. further evaluate the state of the art of using spreading activation for WSD. They state that concepts, which use semantic networks show the best results. Their approach is not capable of 'guessing' better suited concepts than those, which have already been found.

Other approaches to WSD are seen by Agirre et al. ([199]), which use a PageRank based algorithm [200] to disambiguate word senses in the biomedical domain. Kang et al. [201] created a semi-automatic, domain independent approach to WSD (whereas we focus on specific domains). An ontology is created semi-automatically and used for disambiguating the words of a given sentence by finding a least weighted path through the concepts within the ontology. In contrast to our approach they can not resolve reference transfer.

Spreading Activation has been used in other domains as well. Two of these approaches are described in the following. Hussein et al. ([202]) used it for context adaptation. Hence, they model application domains within an ontology. After each user action, an activation flow through the network filters those nodes, which are seemingly most relevant to the current circumstances.

Katifori et al. [203] propose to use spreading activation for personal information manage-

ment in order to provide "context inference to tools that support task information management". They activate concepts which are of importance to the user based on different time scales. These correspond to three different timescales of the human memory [204]. Based on this, their spreading activation infers context elements from an ontology which are most likely to be relevant to the user based on his / her latest set of actions.

6

Semantic Information Retrieval and Classification

6.1. Introduction and Motivation

The analysis process of SE-DSNL delivers precise semantic information, stored within an InterpretationModel. As part of a project scenario the question was asked how SE-DSNL can be used to deliver textual support requests automatically to responsible employees. In order to do so the system must be capable of deciding which InterpretationModel is relevant to which employee. The task can be seen as a problem of classification, i.e., the content of the InterpretationModel must be analyzed according to a set of criteria, all of which are associated to different categories. Depending on the evaluation of those criteria the request is assigned to a corresponding category. For example, a set of criteria can contain a single criterion, which is about engine problems. The criterion is associated to a category 'Engine Development'. If a corresponding textual request would be analyzed, which mentions engine problems, the previous criterion can be evaluated successfully on the request. Accordingly, the request would be sent to the 'Engine Development' department. However, it is sometimes not enough to only classify an InterpretationModel but it is also necessary to extract the information a criterion matches to. This can be seen as an IR task (see section 2.3), i.e., information which conforms to specific conditions must be identified and returned to the user.

In order to solve this challenge we developed a concept which can be used for both classification and information retrieval (it can be best compared to a combination of a SPARQL 'Select' and 'Ask'). The approach is based on so called Patterns. A Pattern specifies a structure between a given set of SemanticElements, which must be part of the InterpretationModel a Pattern is applied to. If a Pattern matches an InterpretationModel we say that the interpretation has been classified according to the specific Pattern. The result of the Pattern application process is that each Pattern element has been assigned to a corresponding InterpretationElement. The approach allows us, e.g., to filter texts which describe a specific scenario in the automobile sector and extract the information which conform to the Pattern. In this chapter we describe the concept and the problems which it has to solve.

The chapter is structured as follows: In section 6.1 we introduce both the problems as well as requirements which we have to consider for the tasks of both classification as well as IR. Next, the structure of the Pattern as well as their inherent semantics are specified in sections 6.2 and 6.3. Section 6.4 presents the algorithm which resolves a Pattern on a given InterpretationModel. Finally, we conclude the chapter by delimiting the approach from existing ones in section 6.5.

6.1.1. Problem Definition

Classical approaches to IR and classification are built upon keyword based concepts, i.e., the requests are parsed for a set of keywords. If, e.g., a specific number of keywords can be found within the request and there is a category which is represented by the same set keywords, the request are classified accordingly. A similar approach for semantic information can, e.g., count the different SemanticElements of a specific type in an InterpretationModel. This already yields some advantages over purely lexical classification or IR approaches, because the SemanticElements in an InterpretationModel are unambiguous in contrast to the words in the text. The approach is straight-forward and simple to realize, however, there are additional advantages if the semantic structure within the interpretation is also taken into consideration. Classifying a semantic interpretation or retrieving information from it therefore bears a set of different problems:

1. InterpretationModels are the result of analyzing natural language text. As humans tend to use multiple ways of describing scenarios, actions, etc., different interpretations for the same scenario may represent different structures. Still, there is a large possibility that some concepts, which are specific to the situation, are almost always identical or at least very similar (note that we are talking about the semantic, not the lexical or syntactic level). They most probably are also related to each other in similar ways, i.e., different InterpretationModels about the same topic contain relations of similar types between the concepts. An example is that one interpretation contains the SemanticElement 'Run' which might be used to relate a person and a location, whereas in another interpretation, instead of 'Run' 'Walk' is the type. Both SemanticElements 'Walk' and 'Run', however, share a very close LCA 'Move' (see section 4.3.2), which indicates a strong similarity between both. The challenge is that in order to analyze and identify such similarities the user should not have to create different criteria for all those interpretations. Instead one criterion for similar interpretations should suffice, i.e., while designing the Patterns the user must be capable of introducing a certain degree of tolerance.

2. In several scenarios it is necessary to know the amount of things, i.e., it must be possible to filter the interpretations based on the number of elements of a specific type. For example, if an author mentions two persons sitting in a car, the Pattern should be capable of identifying only such interpretations which mention at least two persons. This can be difficult in case that more elements of a specific type are contained within the interpretation. The reason is that if more elements exist than what has been specified, the algorithm must be capable of selecting only those SemanticElements which also match the other criteria within a Pattern. As an example imagine an interpretation containing three references to the SemanticElement

'Person', two of which are related to the element 'Car'. The third 'Person', however, has no relation to the 'Car' element. A potential Pattern specifies that the two 'Person' elements must be related to the 'Car'. The algorithm has to identify those two 'Person' elements which also have the relation to the 'Car'. Therefore, it might have to try different combinations of the 'Person' elements in order to find a way how it can solve the Pattern correctly. Aside from this type of cardinality problem there is another challenge which, however, is out of scope of the thesis and is only mentioned for completeness. Humans tend to mention some things and leave out others because they are implicitly known to people who experienced similar situations. Identifying missing information is a very difficult task to handle in a computational domain despite semantic knowledge, as the problem would require very complex knowledge and reasoning about the scenario the user described. Thus it is not in the scope of this thesis.

These two challenges represent the main issues that arise in creating a Pattern based IR and classification concept for InterpretationModels. In the next section we specify the requirements which our approach has to fulfill.

6.1.2. Requirements

In the previous section 6.1.1 two challenges were mentioned which are difficult to handle. Based on these, a set of requirements has been created which our approach has to fulfill in order to sufficiently retrieve information from and classify InterpretationModels.

1. The type of the SemanticElements in an interpretation must be identifiable, i.e., the user must have the possibility of specifying SemanticElements which have to exist within an InterpretationModel. Further the algorithm must be capable of correctly identifying the type as well as take the inheritance hierarchy into consideration while checking the type of an element.
2. The reference for type checking must be the SemanticScope O_{se} (see section 3.3.2 and further) which is also used as an anchor by the InterpretationModels (meaning that the interpretations also reference O_{se} in order to specify the types of their elements). That means that in order to check if an element of the Pattern and an element of the interpretation reference the same type the algorithm must validate if the SemanticElements are related within the SemanticScope.
3. The semantic content of a text is defined by the relations between the SemanticElements of the words. Hence, the algorithm must be capable of classifying interpretations and retrieving information from them which correspond to the type of the relations between the Pattern elements. For example, there are two interpreta-

tions $m_1, m_2 \in O_i$. Both reference the SemanticElements $e_1, e_2 \in O_{se}$ (i.e., there are InterpretationElements which reference the corresponding SemanticElement, but for simplification we only refer to the SemanticElements and leave the InterpretationElements out). m_1 additionally contains an Association a_1 of type $e_3 \in O_{se}$ from e_1 to e_2, whereas m_2 has a different Association a_2 of type e_3 from e_2 to e_1. Classification or IR which is solely based on identifying the correct SemanticElement types, might classify both interpretations as belonging to the same category (because both reference e_1 and e_2). However, if the Associations are also taken into consideration, m_1 and m_2 can be differentiated.

4. Sometimes multiple instances of the same type can be mentioned by a human. Therefore, the number of instances of a single type should be countable, e.g., texts which reference two or more persons which sit in a car. However, it should not be necessary to specify every single instance within a Pattern specification. Instead, it must be possible to define a cardinality for an element, which specifies how many occurrences of the type should be available within an InterpretationModel.

5. Besides validating the existence of specific type occurrences, it is also useful to check for the absence of specific elements. This can refer to both single SemanticElements which should not be contained within an interpretation as well as complete relations which may not exist between elements of a specific type.

6. Humans tendency towards ambiguities also allows for interpretations to consist of different structures despite still containing the same semantic content (as already mentioned in section 6.1.1). This makes classification of as well as IR from interpretations more difficult, because multiple different structures need to be recognized. Hence, the concept must provide a certain tolerance towards specific types of structural as well as semantic variations. One potential concept here is transitivity, i.e., if one element a is related to b and b is related to c, then a is also related to c. Therefore, it might be sufficient to define that a semantically abstract and transitive relation between two elements exists instead of defining its precise type.

7. The algorithm does not have to find the best way a Pattern can match an interpretation. Instead it is sufficient if the Pattern matches the interpretation. That means that as soon as the algorithm can verify that a Pattern completely matches an interpretation the algorithm can terminate. In contrast, however, it has to check every possible combination of elements until there are either no more combinations left or the Pattern can be resolved correctly.

6.2. Pattern Metamodel

Based on the previous requirements, we developed a concept which fulfills the previously specified requirements. The concept is based on the meta model shown in figure 6.1 (the specific semantics of the different attributes are explained in more detail in section 6.3). The elements Scope, ReferencableElement, Element and SemanticElement are known from previous scopes (see chapter 3). Again, there is a specific Scope element: The *PatternScope* element. It contains all *AbstractPatternElement*. There are two specializations of the element: *Pattern* and *SemanticPatternElement*. The first one is the main element of the Scope. It defines what belongs to a Pattern. It therefore references all the semantic information and structures which should be identified within an InterpretationModel. Next, the SemanticPatternElement is used to define the semantic content of a Pattern. The element contains four different attributes: First, *minCardinality* and *maxCardinality* define the minimum and maximum number of elements of a specific type (the type is given via the relation *semanticType*) which have to exist within an interpretation. Next, the attribute *necessity* defines if an element must be found in an interpretation or if it is a 'nice-to-have'. The final attribute *extendedMatching* can be set if the Pattern resolution algorithm should use knowledge from O_{se} to identify an InterpretationElement. The SemanticPatternElement is differentiated in two other elements. The *PatternElement* represents the 'nodes' of a Pattern (a Pattern itself basically represents a graph like structure). The other element *PatternRelationship* is used to model the relations between the different PatternElements. It further contains multiple additional attributes. The first attributes *minTargetCardinality* and *maxTargetCardinality* define a specific type of cardinality, which is explained in section 6.3. The attribute *transitivity* allows the definition of different types of transitivity for a relation.

We can now specify the PatternScope:

Definition 34 (*PatternScope*)

Let

$$O_p := \{p_1, .., p_n\} \tag{6.1}$$

be the PatternScope where $p_1, .., p_n$ are different Patterns within the PatternScope.

The structure of a Pattern is specified as follows:

Definition 35 (*Pattern*)

Let a Pattern $p \in O_p$ be defined as the set

$$p := \{p.E, p.R\} \tag{6.2}$$

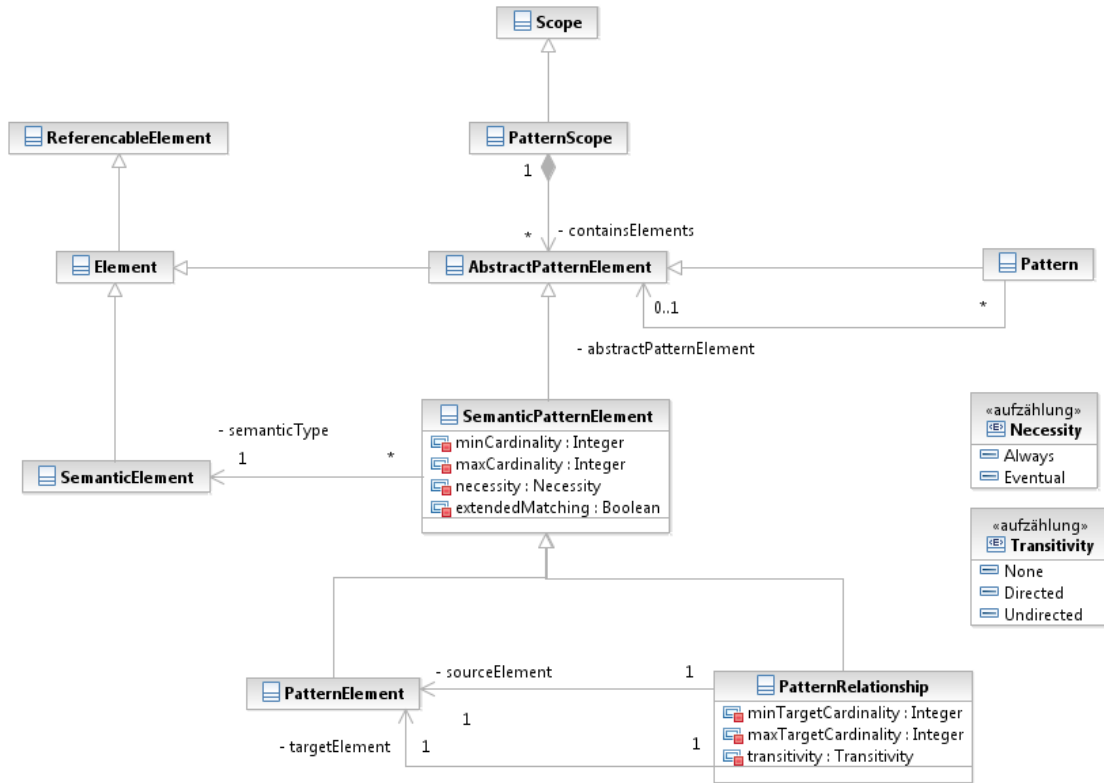


Figure 6.1.: Pattern Metamodel

where $p.E$ is the set containing all PatternElements and $p.R$ is a set containing all PatternRelationships of the Pattern. Further let $p.E \cap p.R = \varnothing$. Each element $e \in p.E \cup p.R$ is defined as

$$e := \{e.sem, e.R_{out}, e.R_{in}, e.minC, e.maxC, e.nec, e.ext\} \tag{6.3}$$

where $e.sem \in O_{se}$ is a link to a SemanticElement in the SemanticScope, $e.R_{out}$ is a set of outgoing PatternRelationships, $e.R_{in}$ is a set of incoming PatternRelationships, $e.minC$ is the same as the previously specified minCardinality attribute, $e.maxC$ equals the maxCardinality attribute, $e.nec$ is the same as the necessity attribute and $e.ext$ equals the extendedMatching attribute. Further all elements $r \in p.R$ contain the attributes

$$\begin{aligned} r := \{&r.sem, r.minC, r.maxC, r.nec, r.ext, r.minTarC, \\ &r.maxTarC, r.trans, r.src, r.trg\} \end{aligned} \tag{6.4}$$

where the first four attributes are the same as specified for the former element e, $r.minTarC$ equals the minTargetCardinality attribute, $r.maxTarC$ equals the maxTargetCardinality and $r.trans$ is the same as the transitivity attribute of the PatternRela-

tionship. $r.src \in p.E$ specifies the source of the relationship, whereas $r.trg$ defines the target of a PatternRelationship.

6.3. Pattern Semantics

The basic semantics are encoded within the structure of the Pattern. The structure is formed by using PatternElements which are connected using PatternRelationships. An example is a Pattern $p \in O_p$ and two PatternElements $e_1, e_2 \in p.E$, which are connected by a PatternRelationship $r \in p.R$. In order to identify the structure within an Interpretation-Model $m \in O_i$, the interpretation has to contain two ConstructionInterpretationElements $i_1, i_2 \in m.E_{con}$ as well as an AssociationInterpretation $a \in m.R_{ass}$ from i_1 to i_2. Then if

$$i_1.sem.sem \subseteq e_1.sem \wedge i_2.sem.sem \subseteq e_2.sem \wedge a.typ.sem \subseteq r.sem \tag{6.5}$$

holds, i.e., if e_1 references the same or a supertype of the SemanticElement of i_1 and e_2 has the same or a supertype of i_2 and the same is true for r and a, p matches m. This means that if a Pattern is applied to an interpretation, the algorithm tries to identify a structure within the interpretation which is similar to the structure defined in the Pattern. The problem of this approach is that interpretations come in many different shapes because of the different ways humans express themselves. Hence, one might need several different and complex Patterns to identify the same scenario in slightly different interpretations. Another possibility is to configure the Pattern such that it contains a certain tolerance. This can be done by using the attributes which have been introduced in section 6.2. Their semantics is explained in the following.

6.3.0.1. Cardinality

In texts often several words may reference the same semantic type, i.e., those words may actually refer to the same real world entity or, at least, they refer to entities of the same type. If, e.g., two ConstructionInterpretations refer to the same entity, there is (in case of a pronominal anaphora) an Association of type 'Equal' between both. As the 'Equal' Association is both symmetric (i.e., for two ConstructionInterpretations $i_1, i_2 \in m.E_{con}$ $i_1.sem = i_2.sem \Rightarrow i_2.sem = i_1.sem$) as well as transitive (i.e., for three ConstructionInterpretations $i_1, i_2, i_3 \in m.E_{con}$ $i_1.sem = i_2.sem \wedge i_2.sem = i_3.sem \Rightarrow i_1.sem = i_3.sem$), ConstructionInterpretations which are connected through an 'Equal' association, all reference the same entity. Such a set is called an equivalence set. However, if there are two ConstructionInterpretations $i_1, i_2 \in m.E_{con}$, which are not connected through an 'Equal' Association, we assume that they reference different real world entities, although both i_1 and i_2 can still reference the same SemanticElement. Therefore, there are two different equivalence sets.

The cardinality attribute defines how many equivalence sets of a specific semantic type (indicated by the $i_1.sem$ attribute of $i_1 \in m.E_{con}$) must be available. The attribute $e.minC$ for

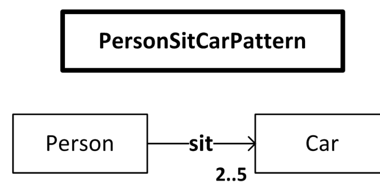


Figure 6.2.: A Pattern, which requires 2..5 persons which sit in a car

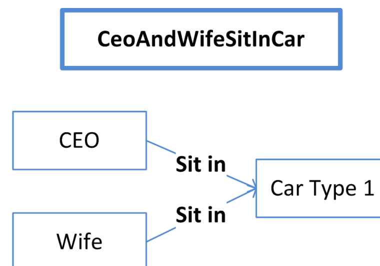


Figure 6.3.: One interpretation matching the Pattern 'Persons Sit In Car'

a PatternElement $e \in p.E \cup p.R$ therefore represents the minimum number of equivalence sets which should be available in an InterpretationModel $m \in O_i$. It represents a hard constraint, i.e., if there are not enough equivalence sets, the minCardinality constraint is violated.

In contrast, the attribute $e.maxC$ for a PatternElement $e \in p.E \cup p.R$ is handled as a soft constraint, i.e., if there are more equivalence sets than there are supposed to be, the matching value of the overall Pattern is lowered for an InterpretationModel $m \in O_i$. This does, however, not completely rule out a Pattern for m. The reason for handling the maxCardinality this way is that in longer texts it is not known if different elements of the same type are later used in a different context, i.e., a context which applies to a completely different Pattern.

A special case occurs if for an element $e \in p.E \cup p.R$ $e.minC = 0 \land e.maxC = 0$. There must not be any element of a specific semantic type within the InterpretationModel.

Figure 6.2 shows a Pattern which requires two to five persons to sit in a car (the car has the cardinality 1..1, i.e., there should be exactly one car in the interpretation). A corresponding interpretation can be seen in figure 6.3 (note that these InterpretationModels are simplifications, i.e., normally not the SemanticElements are connected directly; Instead the ConstructionInterpretations are connected with AssociationInterpretations; Further, each ConstructionInterpretation references one single SemanticElementInterpretation, which specifies the semantic type). Here, a 'CEO' and his wife sit in a specific car. The Pattern in figure 6.2 matches the Interpretation perfectly, because there are two elements of type 'Person' which are both related to the single element of type 'Car, using relations of type 'Sit'.

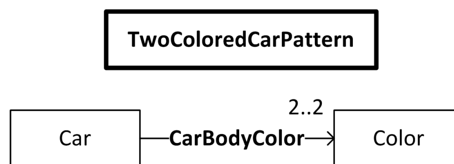


Figure 6.4.: A Pattern, which requires one specific car to have two colors; The cardinality values are values for the minimum and maximum target cardinality attributes

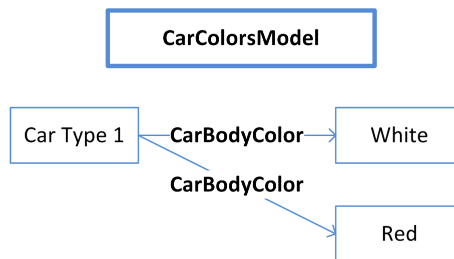


Figure 6.5.: One interpretation matching the Pattern 'Two Colored Car'

6.3.1. Target Cardinality

The target cardinality attributes $r.minTarC$ and $r.maxTarC$ for a PatternRelationship $r \in p.R$ are special cases of the normal cardinality attributes. They define, how many AssociationInterpretations of a specific type must exist per source ConstructionInterpretation. In contrast, the normal cardinality attributes define how many AssociationInterpretations of a specific type must exist in the complete semantic interpretation.

An example can be seen in figure 6.4. The PatternRelationship 'CarBodyColor' has a target cardinality of 2..2 as well as the extended matching attribute set to true (this is described in section 6.3.3), i.e., exactly two colors must be associated to one car. A matching interpretation can be seen in figure 6.5. The car is associated to two colors (the 'State' association matches the 'CarBodyColor' PatternRelationship because of the extended matching attribute). Therefore, the target cardinality of 2..2 can be resolved correctly on the interpretation. If either one of those two relationships are missing, the 'Two Colored Car Pattern' would not match the interpretation.

6.3.2. Necessity

The attribute $e.nec$ of an element $e \in p.E \cup p.R$ specifies if e is a 'must-have' element or not. It can have two values:

1. Always: If the necessity of an element is set to 'Always', the element must always be found within an InterpretationModel in order for a Pattern to match the inter-

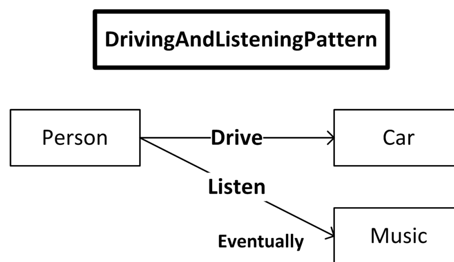


Figure 6.6.: An example for a Pattern, where one relations necessity is set to 'Eventually'

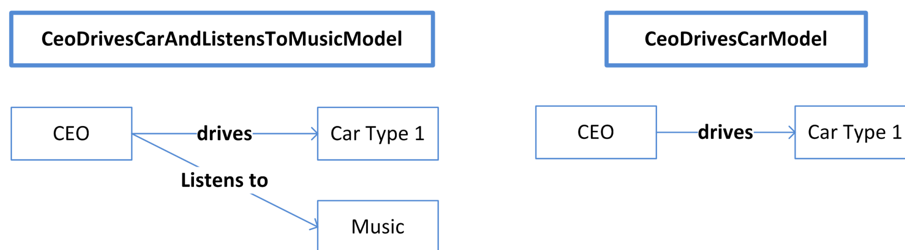


Figure 6.7.: Two interpretations matching the 'Drive and Listening Pattern'

pretation. If the corresponding element is not found, the Pattern does not match the interpretation.

2. Eventually: If the necessity of an element is set to 'Eventually', the element does not need to be found within an InterpretationModel. However, finding the element helps in resolving the corresponding Pattern on a specific semantic interpretation.

The default case is the value 'Always'. An example for the attribute can be seen in figure 6.6. The Pattern simply specifies that a matching interpretation should contain a SemanticElement 'Person', which must be connected to another SemanticElement 'Car'. Additionally, if possible, 'Person' should also be connected to the element 'Music' with a relation of type 'ListenTo'. In figure 6.7 two different interpretations can be seen. The interpretation on the left side matches the Pattern from figure 6.6 completely, i.e., all three elements as well as both PatternRelations can be resolved. The interpretation on the right side, however, misses the 'Listen To' relation. Still, the Pattern 'Driving and Listening' can be resolved on the right interpretation, because the 'Listen To' PatternRelationship has been marked as 'Eventually'.

6.3.3. Extended Matching

The attribute $e.ext$ for an element $e \in p.E \cup p.R$ allows a more tolerant approach to matching the specified semantic type of a ConstructionInterpretation to the semantic type of PatternElement. Figure 6.9 represents four slightly different interpretations. We are going to define a Pattern which only retrieves interpretations in which a car with a red body

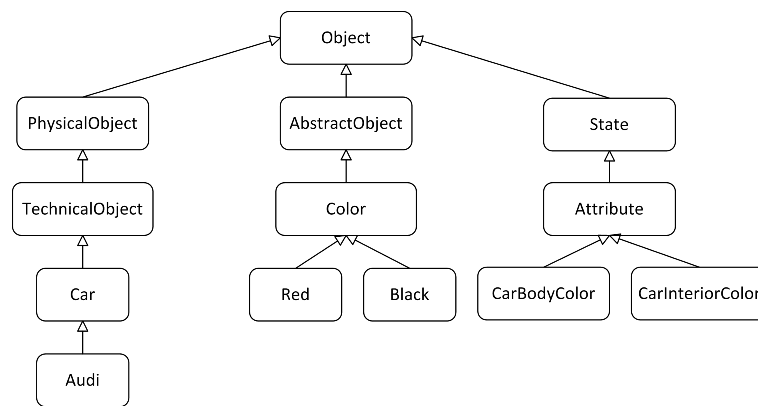


Figure 6.8.: Ontology excerpt for the extended matching example

color is mentioned. Interpretation one contains the correct information. Interpretation two might also be valid, its problem is that the relation between the two ConstructionInterpretations is very generic. As can be seen in the ontology excerpt in figure 6.8, 'State' is a parent class of 'CarBodyColor'. Hence, we don't know, what the author of the text of interpretation two might have meant exactly. We, however, still want to be able to resolve the case (some form of benefit of the doubt). Interpretation three in contrast is about something completely different, as the interior color of the car has been specified (and not the body color). In interpretation four, the body color of the car is mentioned, however, the color is black and not red. Therefore, interpretation four is also not correct.

Potential Patterns which can identify the correct interpretations one and two can be seen in figure 6.10. The Pattern 'CarBodyColorRedPattern' on the left side defines that a 'Car' should be related to 'Red' using a relation of type 'CarBodyColor'. If this is applied to the previous interpretations, it would match only to the first one. The Pattern 'CarStateRedPattern' in contrast matches the interpretations one, two and three. As can be seen, neither of both Patterns can find the correct interpretations only. In this case, the extended matching can be applied. If the attribute is set to true on the relation between the 'Car' and the color 'Red' in the 'CarBodyColorRedPattern', the resolution algorithm does not only search for children of the type of the PatternRelationship element, but also for its parents. Regarding interpretation two of figure 6.9, 'State' is a parent of 'CarBodyColor' (as can be seen in figure 6.8), therefore the Pattern can correctly be applied to it. However, it still does not match to interpretation three (as 'CarInteriorColor' is a sibling of 'CarBodyColor' and not a parent) and four (as 'Black' is a sibling of 'Red').

6.3.4. Transitivity

The concept of transitivity defines that if there are three elements A, B and C and A is connected to B and B is connected to C, A is also connected to C. Transitivity can be

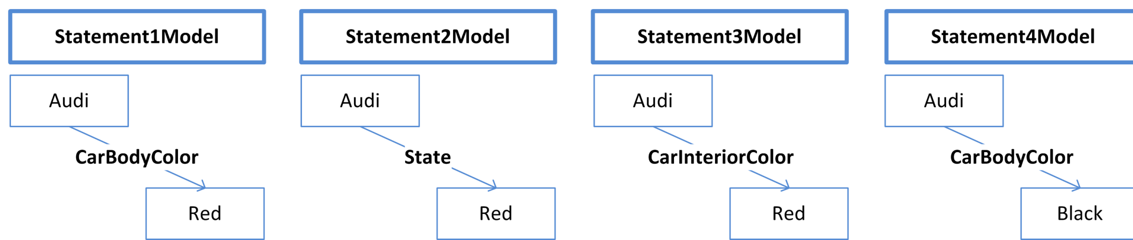


Figure 6.9.: Four similar interpretations for the 'ExtendedMatching' attribute

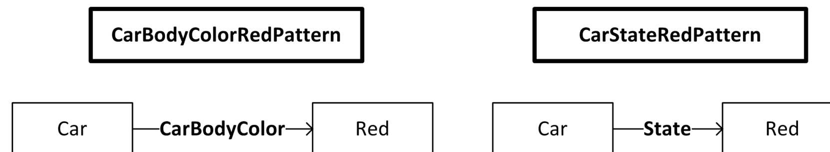


Figure 6.10.: Two Patterns for the 'ExtendedMatching' attribute

helpful for PatternRelationships as it allows a relaxation of a normally strongly fixed structure, i.e., SemanticElements must not be closely related but there could be some additional SemanticElements and relations between them. It can be helpful to identify such types of relations between different elements since humans tend to not only describe the facts and information which they want to communicate, but also accompany the information with additional (sometimes irrelevant) facts and knowledge (e.g., gossip). In order to still identify the correct structures within an interpretation, relations between different concepts must be identified which might contain parts that are not relevant. For a PatternRelationship $r \in p.R$, the attribute $r.trans$ can be set to two different types of transitivity:

1. Directed Transitivity: It is based on the directed AssociationInterpretations between the ConstructionInterpretations. Let us consider the following example: If the ConstructionInterpretations $i_1, i_2, i_3 \in m$ for an InterpretationModel $m \in O_i$ are connected and there is an association from i_1 to i_2 as well as one from i_2 to i_3, then i_1 is also connected to i_3.

2. Undirected Transitivity: In this case, the normally directed AssociationInterpretations in an InterpretationModel are treated as undirected relations. Hence, regarding the previous example with the ConstructionInterpretations i_1, i_2 and i_3, i_3 is also connected to i_2 and i_2 to i_1, therefore i_3 is also connected to i_1. This is in addition to the information which have been generated from the directed transitivity.

It must be mentioned that transitivity always also checks the type of the relations as described previously.

If the maximum cardinality of a transitive PatternRelationship is set to 0, no relation may exist between the source and the target ConstructionInterpretations which should rep-

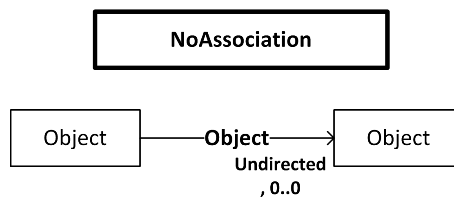


Figure 6.11.: A generic Pattern with an undirected transitivity PatternRelationship

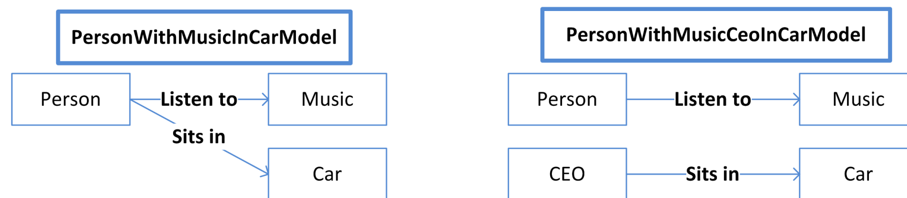


Figure 6.12.: Two interpretations for the 'No Association' Pattern

resent the PatternElements. An example can be seen in figure 6.11, which describes a generic Pattern, where two elements are connected to each other via a generic PatternRelationship. The relationship has its transitivity attribute set to 'Undirected', its cardinality is 0..0 and its type is 'Object' (the most generic element in the ontology). Hence, there should be two InterpretationElements within an InterpretationModel which are in no way connected. Figure 6.12 shows two different interpretations. The interpretation 'PersonWithMusicInCarModel' is fully connected: 'Music' and 'Car' are both associated to the element 'Person'. 'Listen To' is the type of the relation between 'Person' and 'Music' and 'Sit In' is the type of the relation between 'Person' and 'Car'. Therefore, the previous Pattern can not match the model. The other interpretation 'PersonWithMusicCeoInCarModel', however, is not completely coherent, i.e., there are at least two elements which have no relation between them. As can be seen, there is, e.g., no AssociationInterpretation between 'Person' and 'Ceo' or 'Car'. Hence, the Pattern can be applied to the situation: No undirected path of generic SemanticElements can be found between those two elements.

6.4. Pattern Evaluation

The previous section explained how Patterns can be specified. In the following section we describe the algorithm which resolves a Pattern on a given InterpretationModel. The algorithm is separated in two phases. The first phase tries to identify a subset $P_t \subseteq O_p$ of Patterns which seem to be suited for an InterpretationModel. The second phase evaluates each of these Pattern $p \in P_t$ on the interpretation. The reason for the Pattern preselection phase is that the detailed resolution of a Pattern can be a time consuming task, depending on the utilized attributes (e.g., a transitive, but absent PatternRelationship).

In order to explain the algorithm one additional definition is required. It is based on the information which has been presented in the previous sections and is used throughout the remainder of the chapter.

Definition 36 (*Pattern Instance*)

Let

$$p^I := \{p^I.m, p^I.E^I, p^I.R^I, p^I.p\} \tag{6.6}$$

be a Pattern instance of a Pattern $p \in O_p$, which is stored within $p^I.p$. $p^I.m$ specifies the InterpretationModel $m \in O_i$ on which p has been instantiated. $p^I.E^I$ is a table in which every element $e \in p.E$ is linked to a list of possible elements $i \in m.E$. Depending on the cardinality as well as the necessity attributes of each PatternElement $e \in p.E$, there can be zero or more associated InterpretationElements in $p^I.E^I$. Correspondingly, $p^I.R^I$ holds the equivalent information for all PatternRelationships in $p.R$ w.r.t. m.

6.4.1. Phase 1 - Selection

The task of the selection phase is to identify a set $P_t \subseteq O_p$ of Patterns which can match an interpretation $m \in O_i$. A Pattern $p \in O_p$ is a possible candidate for m, if the semantic types of the ConstructionInterpretations in $m.E_{con}$ match the semantic types of the PatternElements in $p.E$. While computing the matches between p and m, a rating is calculated. Based on this value the algorithm decides if p should be evaluated further or not. In the following these mechanisms are described in detail.

Selection of Patterns The selection of Patterns is based on the idea that each PatternElement needs to be instantiated, i.e., only such Patterns can be instantiated whose PatternElements have been assigned to elements from an InterpretationModel. The cor-

responding code can be seen in algorithm 13. The algorithm starts by validating if for every PatternElement $e \in p.E$ there exists an element $i \in m.E_{con}$, whose SemanticElement $i.sem.sem \subseteq e.sem$, i.e., the same as or a child of the PatternElement e (lines 3 to 11). Additionally, it might be the case that the *e.nec* attribute has been set to *true*. In this case, also the other way around has to be checked, i.e., if $e.sem \subseteq i.sem.sem$ (line 7). If in either the first or the second case an element is the child of another one, this fact is stored in the list *patCand* (lines 6 and 8). Next the algorithm validates if for every element, whose *e.nec* attribute has been set to *Always*, there is at least one entry in the *patCand* list (lines 14 to 25). If not, the Pattern can not be applied to the interpretation. Thus, if one element of the Pattern can not be applied to an interpretation m, the Pattern p is not considered for further evaluation.

Algorithm 13 Pattern Selection Algorithm

Input: p: A Pattern
m: An InterpretationModel
Effect: Validates if p matches m; Returns *TRUE* if this is the case

```
1:  procedure MATCHESINTERPRETATION(p, m)
2:      patCand := []
3:      for all e in p.E do
4:          for all i in m.E_con do
5:              if isChild(i.sem.sem, e.sem) then
6:                  patCand := patCand + (e, i)
7:              else if e.ext && isChild(e.sem, i.sem.sem) then
8:                  patCand := patCand + (e, i)
9:              end if
10:         end for
11:     end for
12:     alwaysElements := 0
13:     fulfilledElements := 0
14:     for all e in p.E do
15:         if e.nec = Necessity.Always then
16:             found := false
17:             alwaysElements := alwaysElements + 1
18:             for all t in patCand do
19:                 if t[0] = e then found := true
20:                 end if
21:             end for
22:             if found then fulfilledElements := fulfilledElements + 1
23:             end if
24:         end if
25:     end for
26:     return fulfilledElements = alwaysElements
27: end procedure
```

Rating Patterns, which can be matched to an InterpretationModel in algorithm 13, have to be rated, i.e., for each PatternElement to InterpretationElement mapping (which has been stored within the *patCand* variable) a value is generated. The value represents how semantically similar two mapped elements are. The similarity rating is based on the intersection of semantic information of both elements, i.e., how big the information overlap between the two elements is with respect to O_{se}. The larger the information overlap between two SemanticElements is, the larger their similarity value is, i.e., $sim(s_1, s_1) := 1$ for $s_1 \in O_{se}$. In contrast, the value is almost zero for two totally different SemanticElements (almost, because, as we specified previously in section 3.5.4, there must always be a common root element in O_{se}, therefore two SemanticElements always have an LCA). The algorithm works by counting all parent elements of *e.sem* and *i.sem.sem*. Then the smaller value is divided by the larger value:

$$sim(s_1, s_2) := \frac{MIN(parents(s_1), parents(s_2))}{MAX(parents(s_1), parents(s_2))} \tag{6.7}$$

where $parents(s_1)$ returns the number of parents of $s_1 \in O_{se}$. Note that the function counts each element only once (in case that there is a diamond like generalization hierarchy).

Based on this formula, every set within *patCand* is rated, i.e., the similarity of all sets in *patCand* is summed up and averaged leading to the similarity rating between a Pattern and an InterpretationModel:

$$rating(p, m) := \frac{\sum\limits_{t \in patCand_{p,m}} sim(t.e.sem, t.i.sem.sem)}{\#(patCand)} \tag{6.8}$$

where $patCand_{p,m}$ is the *patCand* table of a Pattern p and the InterpretationModel m, $t.e$ references the PatternElement and $t.i$ specifies the InterpretationElement of a mapping set t. The final step is to select the Patterns for the set P_t. The corresponding code is shown in algorithm 14. It first calculates the average rating for all Patterns matching a given InterpretationModel m (lines 4 to 6). Next, every Pattern which has either an equal or higher rating than the value of *avgRating* or $MIN_{Pattern}$ (a constant threshold), is added to P_t (lines 10 to 12). P_t represents a reasonable subset of Patterns from O_p which are suitable for further evaluation. Next the structure of all Pattern $p \in P_t$ is matched against m.

Algorithm 14 Pattern Rating

Input: O_p: The set containing all Pattern of an SE-DSNL model
m: An InterpretationModel
$MIN_{Pattern}$: Constant threshold

Effect: Returns a set P_t which contains all Patterns that seem to match m

1: **procedure** PATTERNRATING(O_p, m, $MIN_{Pattern}$)
2: $avgRating := 0$
3: $P_t := \{\}$
4: **for all** p in O_p **do**
5: $avgRating := avgRating + rating(p, m)$
6: **end for**
7: $avgRating := avgRating / \#(O_p)$
8: **for all** p in O_p **do**
9: $r := rating(p, m)$
10: **if** $r >= avgRating \,||\, r >= MIN_{Pattern}$ **then**
11: $P_t := P_t + p$
12: **end if**
13: **end for**
14: return P_t
15: **end procedure**

6.4.2. Phase 2 - Evaluation

So far, a set of Patterns P_t has been selected. All of its elements are candidates for matching a given interpretation $m \in O_i$. The algorithm continues by checking which of the Patterns $p \in P_t$ definitely matches m. This is done by validating the structural information of p on m. The problem with this approach is that the analysis can have a high complexity, i.e., there may be several ways how p can be matched to m. The problem is further increased due to several attributes which can be specified on the different elements of a Pattern.

Overall, the structural matching is handled as a search problem, i.e., the algorithm tries to find exactly one way how it can correctly resolve p on m. A Pattern is correctly resolved if, based on its structure and its attributes, the Pattern can be matched on an interpretation, i.e., for all information of the Pattern a corresponding and matching element in the interpretation can be found.

An overview of the algorithm is shown in figure 6.13. Basically, the algorithm starts with an initial state (i.e., S_1 in figure 6.14) and, from there, creates a new state (e.g., S_2). A 'state' corresponds to one step in searching the 'goal', i.e., finding a way to correctly solve a Pattern $p \in P_t$ on an InterpretationModel $m \in O_i$. In each state the algorithm tries to instantiate a single PatternElement from $p.E$, thereby resolving the Pattern iteratively. Next, it inserts new information in the state about how a specific PatternElement can be

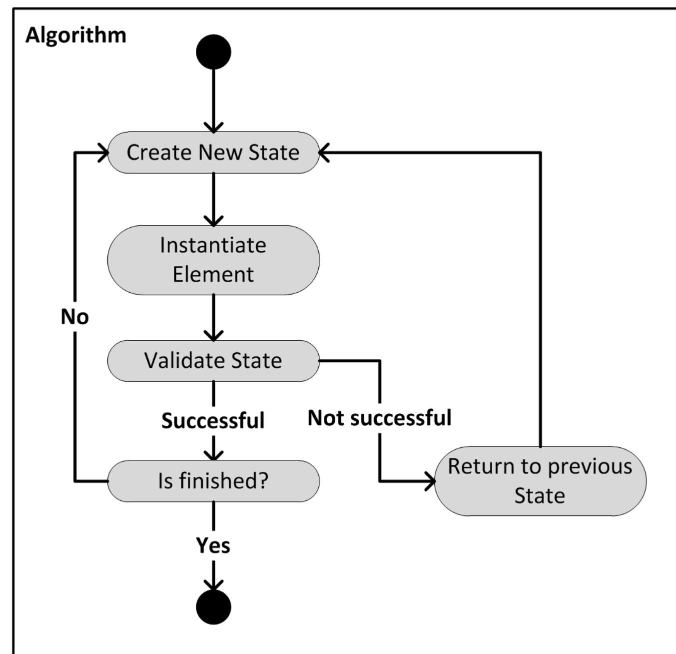


Figure 6.13.: Overview of Phase 2

instantiated with an element from m. The new information is validated, i.e., based on the constraints in the Pattern, the instantiation is checked for its validity. If the validation is not successful, the algorithm returns to the previous state (in the example, this is S_1) and from there on creates a new state (e.g., S_3). The process is being repeated until there are either no more PatternElements available and the last state was validated correctly (in this case the Pattern was resolved successfully) or there are no possibilities left to instantiate a required PatternElement (i.e., one whose necessity attribute was set to 'Always'). In the latter case, p does not match m. However, as soon as one state has been discovered in which all PatternElements have been instantiated correctly, the algorithm terminates. Of course there still can be further and 'better' instantiations of the Pattern, however, it is not the goal to find the best instantiation, but exactly one. In the following, the algorithm is described in detail.

Parsing the Pattern structure In order to start the process, an arbitrary PatternElement e is taken from $p.E$. The element is the initial element for the Pattern parsing process. An example can be seen in figure 6.15. For the example element e_1 has been chosen as the initial element. The algorithm instantiates the PatternElement in the first state S_1. Instantiation means that for e_1 corresponding elements $i \in m.E_{con}$ are searched and 'attached' to e_1. After that, validation takes place. We assume in our example that the elements which have been assigned to e_1, are correct, i.e., the validation of the state is successful. Next, the algorithm analyzes each outgoing relation $r \in e_1.R_{out}$ as well as the corresponding

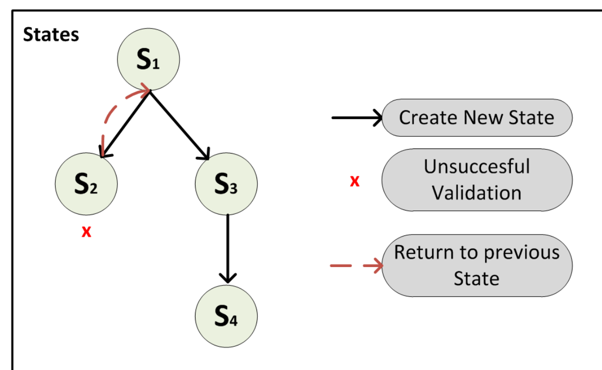


Figure 6.14.: Example of states

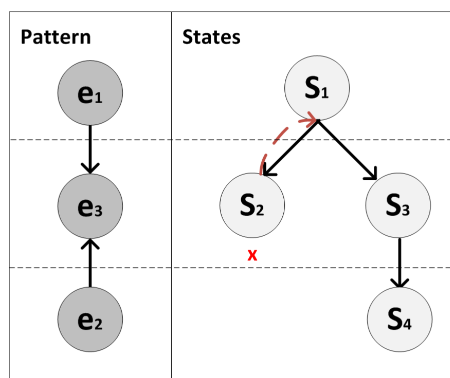


Figure 6.15.: Example of a Pattern structure with the corresponding states

target elements $r.trg$. In the example, there is only one relation outgoing from e_1 to the Pattern element e_3, both of which (the relation and the element e_3) are instantiated in a new state S_2. However, the instantiation can not be validated correctly (indicated by the little red cross), therefore we have to return to the previous state S_1 and search for another state in which e_3 and the relation from e_1 to e_3 can be instantiated correctly. In the example this is S_3. However, another problem occurs. Yet not all elements of the Pattern have been instantiated: The relation between e_3 and e_2 as well as e_2 itself are missing. The problem is that there is no relation outgoing from e_3 which could lead to e_2. Therefore, the algorithm starts backtracking, i.e., every time a new element is reached also all of its incoming relations $e.R_{in}$ as well as the elements on the other side of the relation are checked. Hence, in figure 6.15 there is a new state S_4 which holds an instantiation for the Pattern element e_2 as well as the relation from e_2 to e_3. We assume that the instantiation in S_4 is valid, therefore the Pattern has been resolved correctly on a given interpretation. Next we show how exactly the instantiation as well as the validation of a state works.

Instantiation and Validation of a Pattern Element In order to instantiate a Pattern p on an InterpretationModel m, we first define the PatternElement instantiation itself:

Definition 37 (*PatternElement Instantiation*)

Let $p \in P$ be a pattern and $m \in O_i$ be an InterpretationModel. The instantiation function is defined as

$$\theta : p.E \rightarrow \mathcal{P}(m.E_{con}) \tag{6.9}$$

where θ returns a set of ConstructionInterpretations for a given PatternElement.

In order for an instantiation to be successful, two conditions have to be met:

1. For a given PatternElement $e \in p.E$ and its semantic type $e.sem$ there must be a set $E'_{con} \subseteq m.E_{con}$, such that

$$\forall i \in E'_{con}\ i.sem.sem \subseteq e.sem \tag{6.10}$$

   i.e., every potential instance i of a PatternElement e must be of a semantic type which is either the same as or a child of *e.sem* (in case that *e.ext* is *true*, *e.sem* can also be either a child of or equal to *i.sem*).

2. Every ConstructionInterpretation can only be used in one instantiation, i.e., there must not be two or more PatternElements which contain the same ConstructionInterpretation in their set of instances:

$$\forall e_1 \in p.E\ \neg\exists e_2 \in p.E\ \neg equal(e_1, e_2) \wedge overlap(\theta(e_1), \theta(e_2)) \tag{6.11}$$

   where $equal(e_1, e_2) \Leftrightarrow e_1 = e_2$, i.e., it checks if two PatternElements are identical and $overlap(\theta(e_1), \theta(e_2)) \Leftrightarrow \theta(e_1) \cap \theta(e_2) \neq \emptyset$, i.e., it returns *true* if the intersection of two sets is not empty.

The process of instantiation itself is simple. The algorithm searches for elements within m which match the two prior conditions given a PatternElement $e \in p.E$. The algorithm tries to instantiate it with at least as many elements as the cardinality attributes *e.minC* and *e.maxC* specify (and, in case of PatternRelationships, also the target cardinality attributes). If there are not enough matching elements in m available, the instantiation fails. However, if there is a set $I_{mat} \subseteq m.E_{con}$, which matches a PatternElement $e \in p.E$, and the number of elements in I_{mat} is larger than the value in *e.maxC*, a subset $I'_{mat} \subseteq I_{mat}$ must be chosen. Selecting the elements for I'_{mat} is based on the ratings which have been generated as described in section 6.4.1. Additionally, elements $i \in I_{mat}$ which have relations of the same type as the PatternRelationships of the current PatternElement e are being preferred over those which do not. Hence, an element $i \in I_{mat}$, which is an instance of a PatterElement $e \in p.E$, is preferred in the process of creating the subset I'_{mat}, if at least one

of its relation $r \in i.R_{ass}$ can be used to instantiate one of the relations in $e.R_{out}$:

$$\exists r_e \in e.R_{out} \; \exists r_i \in i.R_{ass} \; r_i.typ.sem.sem \subseteq r_e.sem \tag{6.12}$$

Hence, if $\#(I_{mat}) > e.maxC$, the number of possible states regarding the instantiation of the PatternElement e is

$$\frac{\#(I_{mat})!}{(\#(I_{mat}) - e.maxC)! \; e.maxC!} \tag{6.13}$$

However as the algorithm terminates as soon as it found one solution for the complete Pattern it is rarely the case that all those combinations have to be tried by the algorithm.

The validation of an instantiation depends on the PatternRelationships. Until now the only thing which was validated has been the cardinality attribute. In case that there are not as many elements in the interpretation available as the *e.minC* attribute specifies, the validation of the current state fails. If, however, the validation was done successfully, the instantiated PatternRelationships have to be validated. There are basically four different semantics of how a PatternRelationship $r \in p.R$ can be specified. In the following their semantics are explained and when validation fails.

1. Standard Matching: The standard matching process means $r.trans := None$ and $e.maxC > 0$, i.e., no transitivity is set and the maximum cardinality is greater than zero. The first thing, which has to be verified in this case, is that there are enough AssociationInterpretations available in m which are of the same type as the corresponding PatternRelationship r. Next, the semantic type of the source and target PatternElement of r must correspond to the source and target elements of the AssociationInterpretations. If this is not the case, the instantiation fails.

2. Empty Matching: The empty matching occurs if $e.maxC = 0$, i.e., the maximum cardinality of a PatternElement or PatternRelationship is set to zero. This represents that the information must not exist within an interpretation m. Hence, the algorithm has to verify the absence of the element, i.e., if there are two PatternElements $e_1, e_2 \in p.E$ as well as a relation $r \in p.R$ with $r.src := e_1$, $r.trg := e_2$ and $r.maxC = 0$, there must not exist a structure within m such that any instances $i_s \in \theta(e_1)$ and $i_t \in \theta(e_2)$ are connected by a relation $i_r \in \theta(r)$. In other words: There must not be a relation of the same type as r from i_s to i_t. If this is the case, the instantiation fails. Note that this is a special case of instantiation, because the algorithm tries to identify a missing element, therefore the PatternElement whose absence is validated can not be instantiated. In the previous example the result of $\theta(r)$ is therefore an empty set in a successfully validated state.

3. Transitive Matching: Transitivity specifies that if one element e_1 is connected to an-

other element e_2 and e_2 is further connected to another element e_3, then e_1 is also connected to e_3. During the structural matching, the type of the relations between elements is relevant, even for transitive PatternRelationships. Therefore, if e_1 is connected to e_2 via a relation of type t_1 and e_2 is also connected to e_3 via the same relation type t_1, then e_1 is also connected to e_3 via a relation of t_1. However, if e_2 is connected to e_3 with a relation of type t_2, it would become difficult to specify if e_1 and e_3 are still connected through transitivity. In this case the type of a transitive PatternRelationship r is used to validate this situation. If both $t_1 \subseteq r.sem \land t_2 \subseteq r.sem$ (i.e., both t_1 and t_2 are children of the element $r.sem$), e_1 is connected to e_3 via the type of p_r. If, however, no chain of AssociationInterpretations between e_1 and e_3 can be found which matches $r.sem$, there is no transitive relation.
The transitivity attribute can further be either 'directed' or 'undirected', e.g., the previous example was a directed transitivity. Undirected transitivity means that if t_1 and t_2 are children of $r.sem$, then e_2 is connected to e_1 and, further, e_3 to e_2. This leads to e_3 transitively being connected to e_1. Also, the 'Empty Matching' semantics, as explained previously, can be applied to the transitivity check, i.e., if the maximum cardinality is set to zero the algorithm tries to show the absence of any relation chain which matches the type of the corresponding PatternRelationship between two elements.

4. Domain Knowledge Matching: In case that neither of the previous cases can be applied, the algorithm returns to a default, i.e., it searches for information within O_{se}which can match the PatternRelationship. For example, in a Pattern p there are the PatternElements $e_1, e_2 \in p.E$ as well as a PatternRelationship r from e_1 to e_2. Further, the interpretation m contains an element $i_s \in \theta(e_1)$. However, there is no relation available from i_s which matches the type $r.sem$. In this case, the algorithm analyzes if the SemanticElement $i_s.sem.sem$ has an Association of the same type as $r.sem$ within O_{se} and the target of the Association is of the same type as e_2. If this is the case, the relation can again be instantiated successfully.

Every PatternRelationship, which is instantiated in a new state, has to fulfill its corresponding criteria. If it can not, the current state is not valid and the algorithm returns to the parent of the current state. However, if the instantiation is valid, the state can itself act as a parent for new states. The process of instantiation is repeated until one state is found which successfully fulfills the complete Pattern. In contrast, if no single state can be found which successfully fulfills the complete Pattern, the Pattern can not be resolved successfully.

6.5. Related Work

Our approach is similar to such which try to retrieve information in OBIR based systems. Therefore, the focus is on the relation of OBIR system and our approach. Some of them were already discussed in section 4.4.

A common way for information retrieval from ontologies is RDQL [205] and SPARQL [206], which allow querying RDF graphs. In the following, we only describe SPARQL, since RDQL is a subset of SPARQL. It supports different types of queries:

1. SELECT queries variables which have been specified in a WHERE clause.
2. CONSTRUCT returns an RDF graph as the result of a specified graph template.
3. ASK returns just true or false given a specific query pattern.
4. DESCRIBE extracts an RDF graph, however, the resulting RDF graph is determined by the SPARL endpoint, i.e., the system which 'answers' the query request.

A Pattern can be seen as a mixture of a SELECT and an ASK SPARQL query. If the Pattern can be resolved correctly, this corresponds to an ASK query which returns true. Every PatternElement is bound to specific elements within an InterpretationModel, thereby retrieving information similar to a SELECT query. If, however, a Pattern is not resolved correctly, this equals an ASK query which returns false. Regarding the expressiveness of the query, the available variables of a SPARQL query correspond to SemanticPatternElements, the WHERE clause of a SPARQL query is best compared to the PatternRelationships as well as the different attributes of a SemanticPatternElement. The main difference between both approaches is the domain in which they are being applied. SPARQL is used directly on RDF graphs which contain facts only, whereas Patterns can be used to identify specific structures within an InterpretationModel of natural language text. Depending on the domain and therefore how the InterpretationModels are constructed (i.e., how the Constructions have been designed), Patterns and their structure must also be adaptable to the domain as well. The official SPARQL standard does not support things like transitive properties, however, there are modified implementations of SPARQL available which provide such functionality (e.g., ARQ [207] or Virtuoso [208]). Other features like specifying the cardinality of an PatternElement are also not available within SPARQL, but are supported to a certain degree in ARQ [209]. A difference between the SE-DSNL Pattern approach and ARQ is that ARQ works on a complete RDF graph. In contrast, an InterpretationModel is not itself part of the SemanticScope, but a separate scope which references elements within the SemanticScope. Therefore, a Pattern can either query the information within the InterpretationModel only or can be specified such that it further accesses information within the SemanticScope.

There are other approaches for information retrieval from ontologies. One interesting approach was taken by Toma [148]. He constructs a so called query ontology o_s from either a query interface or an agent. The query ontology is matched against a set of ontologies $o_1..o_n$ which have previously been matched against a set of documents (however these have not been extracted from the documents). The ontologies $o_1..o_n$ contain factual information in contrast to the InterpretationModel. The matching between o_s and $o_1..o_n$ is based on a similarity measure by Niwattanakul et al. [210] and the Jaccard coefficient [211]. However, it is unclear, to which degree this approach considers structures or attributes like transitivity in matching the query ontology to other ontologies.

Ruotsalo [212] developed an information retrieval system based on the vector space model [213], which uses RDF triples instead of simple concept detection as it was often done in other approaches [214] [215] [216] [136] [217] [135] [146] [218]. Because of RDF, however, it does not support more advanced features like transitivity or cardinality attributes.

7

Evaluation

7.1. Introduction and Motivation

In order to evaluate the theoretical ideas of this thesis, we developed a prototypical implementation[1] which should demonstrate the feasibility of our theoretical concept. In the following, we describe the prototypical implementation and demonstrate with two case studies that the theoretical concepts work the way that they were intended. The evaluation is based on two parts: A descriptive as well as an observational evaluation (as it is proposed by Hevner et al. [219]). The descriptive evaluation (section 7.3) is based on scenarios and separated into three parts ([220]):

1. Modifiability: We show the impact that different requirements for modifying knowledge can have on the ontology. Further we argue how much effort is required to fulfill these requirements.
2. Reusability: Certain scenarios can be solved much faster if knowledge or components from previous projects are reused. We present different scenarios and how reuse does affect them.
3. Performance: An important part of any application is its performance. Therefore, we argue what the most critical part of the SE-DSNL concept is and which parameter has the biggest impact on performance.

To show that the overall concept works as intended we created two different case studies:

1. The first case study focuses on the question if and how SE-DSNL can be applied to linguistically broad domains. We further investigated if the integration of semantic knowledge into the analysis process is helpful. For this, several smaller texts were randomly selected from an online car community (amongst others), for which an ontology was built. Next the tests were analyzed by the SE-DSNL prototype and the results were checked for their validity.
2. The second case study tried to answer the question if SE-DSNL can be used in a real world application in order to control the application with natural language commands. The difference between both case studies is that the linguistic expressiveness in this case study was limited to a bare minimum by using a simple controlled natural language. Based on this, a set of commands was created which had to be parsed with maximum precision and an acceptable performance. The result of the parsing process had to be classified according to the function that the command should actually execute. Therefore, the Pattern concept from chapter 6 was applied. Further the SemanticElements which should be used as input parameters

[1] Download: http://thesis.ds-lab.org/wolffischer/

for the to-be-executed-function were retrieved from the InterpretationModels.

The chapter is structured as follows. First, section 7.2 gives an overview of the prototypical implementation. Next, we discuss the different scenarios in section 7.3. The first and second case study follow in sections 7.4 and 7.5.

7.2. Framework Architecture

This section gives an overview of the architecture and all components which have been developed as part of the thesis. Further the prototypical implementation is explained. Figure 7.1 shows the main components of our framework. At its core, there is the *sedsnl::ontology* component which is a representation of the meta model described in section 3. The component automatically manages the correct creation, altering and storage of ontologies. Each other component is based on it. In order to create a SE-DSNL model the *sedsnl::design* component provides an application which supports an expert in this task. While creating or altering a model, the *sedsnl:validation* component validates that the information in the model confirms to our guidelines of section 3.5.4. The validation is based on the data-flow framework [113]. The *sedsnl::design* component further uses the *sedsnl::owlImport* component which allows the direct import of information from an existing OWL ontology (as described in section 3.5.1). Three different components have been developed to allow a simplified ontology design by focusing only on the relevant information. First, *sedsnl::filterView* allows the filtering of information in a SE-DSNL model by searching for specific information as well as hiding irrelevant information. Next, *sedsnl::constructionView* provides a simplified way of designing Constructions, i.e., an expert using this component receives suggestions to what he might have meant (this is similar to modern programming environments which suggest to the programmer which methods he / she can use within a given class). The component *sedsnl::mappingView* provides a simplified way of associating SemanticElements to their corresponding Forms. The *sedsnl::analysis* component analyzes text based on the information available within a SE-DSNL model, which is provided by the *sedsnl::ontology* component. The analysis component further makes use of the features offered by the *sedsnl::pattern* component. It realizes the functionality which has been described in chapter 6. The syntax parser which is used for the analysis component is provided by the *StanfordParser* component. A simple web interface for the *sedsnl::analysis* component has been implemented in the *sedsnl::webInterface* component.

Figure 7.2 gives an overview of the core classes of both the *sedsnl::analysis* (blue entangled) and *sedsnl::pattern* (green entangled) components. The starting class is the *AnalysisPipeline* class which controls the overall process by calling all different components when they are needed. The process begins in the *SemanticTextSolver* class. It initiates the syntax parser (i.e., the Stanford Parser in *StanfordTreeNode*). The root node of the syntax tree is fed to the *PostOrderRunner* class. It checks the different nodes of the tree and accordingly calls the *ComplexConstructionQuery* class which applies Constructions. It, therefore, executes the specified Statements (only a few can be seen in the figure, i.e., *InOrder*, *CheckForTriple* and *FindAnaphoras*; The latter two further call the *SemanticSpreadingActivation* class,

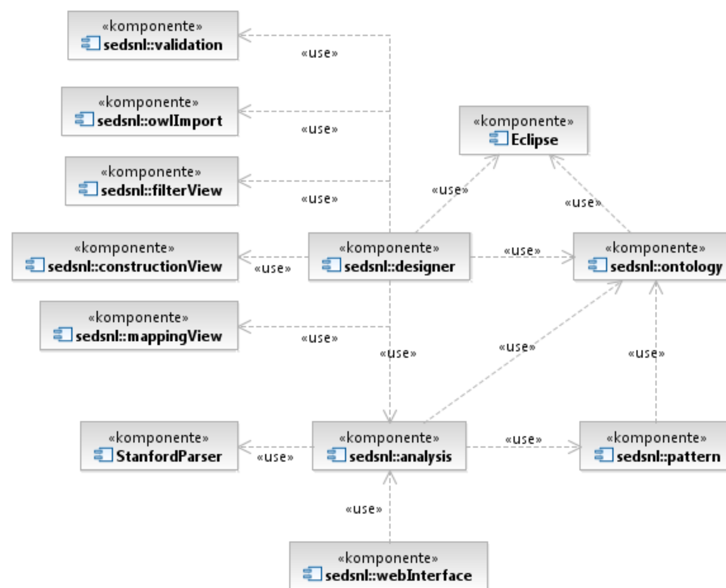


Figure 7.1.: Framework architecture

which implements the functionality of section 5). After the *ComplexConstructionQuery* class finishes the computation on a *SyntaxTreeNode*, all *ConstructionInstance*s are added to the *InstanceManager*. It first gives all new *ConstructionInstance*s to the *InstanceClassification*, which classifies and filters new information. Next, the *InstanceManager* stores the information. If the overall process has terminated, the solutions are extracted and stored inside the *TextSolutionPath*s. This serves as the basis for the *InterpretationCreator* which creates the InterpretationModels. This is the input for the *PatternEvaluation* which takes these InterpretationModels and first tries to map all available Patterns to them in the *PatternCombination* class. The result is stored in the *PatternElementCandidate* class and given to the *PatternSolver* which evaluates both the structural relations as well as attributes of a Pattern. For the process it uses the *DecisionEngine* which allows the algorithm to return to any previously made decision and search for other possible outcomes.

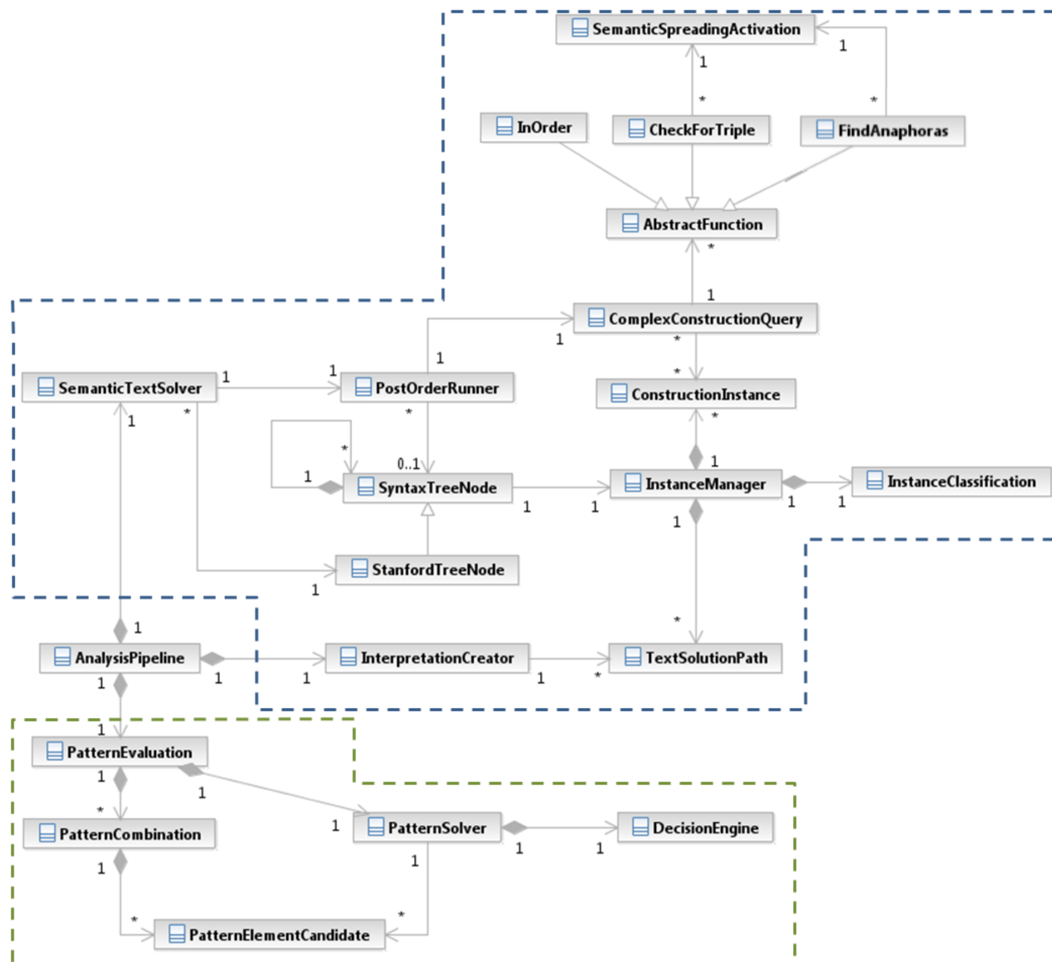


Figure 7.2.: Class diagram of the core classes of the Analysis and Pattern components

7.3. Scenario-based Evaluation

To study quality related aspects of a concept or prototype the method of creating and evaluating scenarios has been established. A scenario is described as "a set of situations of common characteristics that might reasonably occur in the interactions between stakeholders and a system" [220]. In the following we describe several different situations which might occur during the lifetime of the SE-DSNL concept and its prototypical implementation. We show how these situations can be handled and how much effort is needed in order to cope with the specific situations. There are two main sections: The first one 7.3.1 describes different kinds of scenarios all of which are related to modifying information as well as specific components of the SE-DSNL prototype or an ontology. The second section 7.3.2 defines scenarios which might occur if someone is trying to reuse specific parts of the SE-DSNL prototype or ontology.

7.3.1. Modifiability

In this section we describe a set of scenarios which describe situations regarding the modifiability of the software. This basically means scenarios in which parts of our prototypical implementation have to be adapted in order to meet new domain specific requirements. Each of those scenarios is rated on how difficult they are to accomplish, i.e., the amount of work needed to realize them.

Scenario 1: Updating lexical information In any domain, new descriptions and terms can be introduced or existing ones might be changed. Adding new lexical information to SemanticElements is simple, as it is facilitated by the *sedsnl::mappingView* component (see figure 7.1). It, however, is more difficult if compound terms have to be mapped as this might require structural information. Therefore, mapping compounds requires more complex Constructions which have to be created within the *sedsnl::constructionView* component. Most of the time, however, adding new lexical information or changing existing ones is only related to atomic forms, which is a fast process. Table 7.1 gives an overview of the different aspects.

Scenario 2: Editing semantic information The knowledge of domains changes constantly. The semantic information which represents the domain must, therefore, also be updated continuously. The *sedsnl::design* component together with the *sedsnl::filterView* component facilitates the work of an expert, i.e., editing the available information or adding new information. However, it must be noted that adding new semantic information might also require the mapping of new SemanticElements to existing or new lexical

Table 7.1.: Scenario 1: Updating lexical information

Aspects	Description
Causes	- Introduction of new Forms for existing SemanticElements - Editing existing Forms
Goals	- The new lexical information has been mapped to the SE-DSNL model - The lexical information has been altered
Quality Attributes	
Approach	Use the *sedsnl::design* component to add, edit and map lexical information
Risks	No known risks
Sensitivity Points	Compound terms are more difficult to treat than atomic forms; Creating and mapping such is facilitated by the *sedsnl::constructionView* component
Effort	Low, because process is facilitated by components

information as in the previous scenario. Depending on how much information has be changed or added, the process may be time consuming but is easy to handle. The difficulty lies in keeping the ontology up to date with the real world information of the domain. Table 7.2 gives an overview of the different aspects.

Table 7.2.: Scenario 2: Editing semantic information

Aspects	Description
Causes	- Introduction of new semantic information - Update of existing information required
Goals	- New semantic information have been integrated into a SE-DSNL model - Existing semantic information have been updated
Quality Attributes	
Approach	Use the *sedsnl::design* component to add, edit and map semantic information
Risks	- New information might eventually have an impact on the result of some of the algorithms
Sensitivity Points	- Mappings to lexical information might be required - Continuously changing domains require continuous changes to the SE-DSNL model
Effort	Low, because process is facilitated by components

Scenario 3: Modifying Functions New languages, new types of information or new algorithms might require new Functions such that a new language can be parsed or that new types of semantic knowledge can be incorporated into the parsing process. Adding

new Functions is a straight forward process. Basically, the information about the new Function has to be inserted into the ontology by adding a new Function element to the ConstructionScope. Next, a new class with the same name as the Function must be created as a child of *AbstractFunction* (seen in figure 7.1) within the *sedsnl::analysis* component. The new Function is ready and can be used by Statements as soon as the *sedsnl::analysis* component has been recompiled. The process is neither difficult nor time consuming. Table 7.3 gives an overview of the different aspects.

Table 7.3.: Scenario 3: Modifying Functions

Aspects	Description
Causes	New functionality because of , e.g., a new language is required
Goals	A new Function, which has been integrated into the *sedsnl::analysis* component
Quality Attributes	Depending on the Function, Performance and Stability can be affected
Approach	- Use the *sedsnl::design* component to update the required Constructions - Implement and integrate the new Function into the *sedsnl::analysis* component
Risks	- Unexpected results of the analysis process - Eventually lowered runtime performance or stability
Sensitivity Points	
Effort	Low, if the functionality is already available

Scenario 4: Updating Constructions Parsing grammatical structures is based solely on the Constructions within the ConstructionScope. If new Constructions have to be created or existing ones should be edited, this can be done in the *sedsnl::constructionView* component. Depending on the overall complexity and amount of grammatical structures to parse, creating Constructions can be both difficult and time consuming. The reason is that an expert which creates the Constructions, directly influences the precision, tolerance and performance of the *sedsnl::analysis* component. If the Constructions are too generic, many new Construction instances might be created at runtime, leading to a decrease in performance and perhaps even precision. However, a too precise Construction might only be applicable to very specific grammatical structures, which in turn might require additional Constructions for similar, but not identical structures. Finding the correct mixture is an iterative process. It, therefore, requires an experienced user with knowledge about the current domain. The expert also needs knowledge about the design of Constructions. Table 7.4 gives an overview of the different aspects.

Table 7.4.: Scenario 4: Updating Constructions

Aspects	Description
Causes	- New grammatical structures have to be parsed - Existing parsing results should be restructured
Goals	Newly created or updated Constructions within the SE-DSNL model
Quality Attributes	Runtime performance and precision can be affected
Approach	Use the *sedsnl::constructionView* component to create or update a (new) Construction
Risks	- Unexpected results of the analysis process - Strong correlation with runtime performance
Sensitivity Points	Complex Constructions have the highest impact on runtime performance
Effort	Medium to High

Scenario 5: Changing the syntax parser Changing the syntax parser can be a difficult task to do and depends on different parameters. The main question is how much the new parser component differs from the previous one both in terms of the quality of the result as well as the names of the different POS and syntactic categories. In order to change the parser component basically two steps have to be done: First, the existing parser has to be exchanged from an architectural point of view. This means for our prototypical implementation that the Stanford Parser would be exchanged with a different parser. The architectural impact is low as only one class has to be edited. This can be seen in figure 7.2. Only *StanfordTreeNode* has a reference on the Stanford Parser. Hence, a sibling class to *StanfordTreeNode* should be created, implementing all the features which are required to correctly instantiate and execute the new parser component. Further it must be registered in the *SemanticTextSolver* class. This is enough from an architectural point of view. A bigger problem is imposed by the SyntacticCategories of the current SE-DSNL model. They are used to map the different labels of the syntax tree to the ontology. Different parsers may provide at least some different tags, even for the same language. Therefore, the mapping between the syntax tree nodes and the SE-DSNL model would be incomplete, meaning that the user must manually update the SyntacticCategories as well as all Constructions which use outdated information. The process can be facilitated with proper tool support. Further, depending on the way that the new parser creates its syntax trees, all Constructions might have to be adapted to the new grammatical structures as well. This is a very time consuming process (as already described in the previous scenario). It might actually be easier to start from scratch in this case. Overall it is a very difficult scenario to handle. Table 7.5 gives an overview of the different aspects.

Table 7.5.: Scenario 5: Changing the syntax parser

Aspects	Description
Causes	- A new language must be parsed - A new parser offers better precision or performance
Goals	Fully working SE-DSNL framework with the newly integrated syntactic parsers
Quality Attributes	Performance, Precision and Stability
Approach	- Exchange the syntactic parser component in the SE-DSNL component - Update the SE-DSNL model
Risks	- Unexpected results of the analysis process - Unexpected behavior of the analysis process
Sensitivity Points	- New SyntacticCategories as well as new taxonomy required - Existing Constructions might have to be updated
Effort	High

Scenario 6: Adding a new Language The time it takes to represent a given SemanticScope in a complete new language depends on many different factors. This basically is a collection of all the previous scenarios. First of all, all the SemanticElements must be represented with terms of the new language. This is a straight forward process as it as already been described in the previous scenario 1. Next, a new syntactic parser might be needed. The architectural implementation is straight forward. Additionally, as we are not coping with existing Constructions, the expert can start from scratch, which might actually be easier than updating existing ones. If the new Constructions require new Functions, those have to be created as well (see scenario 3 for more details). Overall, adding a complete new language is the most time consuming and difficult scenario as it incorporates every previous scenario and, therefore, all the steps which are required to fulfill the prior scenarios. An overview of the different aspects of the scenario is given in table 7.6.

7.3.2. Reusability

Existing information or components should be reused if possible. As has been shown in the previous section, a knowledge intensive concept can be difficult and time consuming to maintain. Especially language related changes can lead to problems. However, specific parts can also be reused and do, therefore, not occur very often.

Table 7.6.: Scenario 6: Adding a new Language

Aspects	Description
Causes	A new language must be parsed
Goals	Semantic information within a SE-DSNL model has been mapped to a new language
Quality Attributes	Performance, Precision and Stability
Approach	- Exchange the syntactic parser component in the SE-DSNL component - Update the SE-DSNL model
Risks	- Unexpected results of the analysis process - Unexpected behavior of the analysis process
Sensitivity Points	Similar to all previous scenarios
Effort	High

Scenario 7: Introducing the SE-DSNL concept to a new but known domain Creating all the information from scratch (i.e., SemanticScope, SyntacticScope and Construction-Scope) can be a very time consuming task. However, if an SE-DSNL model has been created for a similar domain previously, parts of it can be reused in other projects. The SemanticScope can for example be reused, at least specific parts of it (i.e., the topmost part). This facilitates the overall process of introducing the new technology to an unknown domain and thereby reduces the required time. It is especially helpful, if the new domain requires the same language as the previous one. In this case, even the lexical representations as well as Constructions and Functions can be reused (if the syntax parser also stays the same). The latter especially is the most time consuming task in introducing the SE-DSNL concept. Hence, if the linguistic information has been modeled once, it can and should be reused. An overview of the scenario is available in table 7.7.

Table 7.7.: Scenario 7: Introducing the SE-DSNL concept to a new but known domain

Aspects	Description
Causes	Available Domain knowledge must be updated and mapped to natural language
Goals	SE-DSNL has been fully integrated into a new domain
Quality Attributes	Security can be an issue, depending on the owners of the domain knowledge
Approach	- Introduce the existing SE-DSNL model - Update it such that the new domain is fully represented
Risks	Security
Sensitivity Points	The time to introduce SE-DSNL depends on the similarity of both domains
Effort	Medium to high

Scenario 8: Introducing the SE-DSNL concept to a new and unknown domain Introducing SE-DSNL to a completely unknown domain is the worst case possible regarding the required time and effort. Because the domain is completely unknown, there is no prior information available, neither semantic nor linguistic knowledge. This means that there is no reuse possible. As has been elaborated in the previous scenarios, creating linguistic knowledge can be very time consuming (depending on the grammatical complexity). Further, architectural changes or new Functions might be required. Basically, everything has be created from scratch. This is why it is the worst case possible. An overview can be seen in table 7.8.

Table 7.8.: Scenario 8: Introducing the SE-DSNL concept to a new and unknown domain

Aspects	Description
Causes	Domain knowledge must be created and mapped to (unknown) natural language
Goals	SE-DSNL is fully integrated into the domain
Quality Attributes	Performance, Stability
Approach	- Create a new SE-DSNL model and all its scopes - Create new Functions and integrate them into the analysis process
Risks	Required time is difficult to estimate
Sensitivity Points	Many tests are required in order to achieve good parsing results
Effort	High

Scenario 9: Portability In the computer industry, several different platforms exist. The question is how much effort is required to port the current prototype to other platforms. The prototype has been developed based on Java and Eclipse only. Therefore, porting the code should be simple as many platforms offer Java and Eclipse support. This makes it possible to port the code to platforms like MacOS or Linux without big changes to the SE-DSNL framework prototype. However, if either Java or Eclipse support or both are missing, there are still ways of enabling access to the *sedsnl::analysis* component at least, e.g., by wrapping the prototype in a web service (which has already been done in component *sedsnl::webInterface* as seen in figure 7.1). Hence, the prototype can be ported easily by reusing most of the existing code and components. Table 7.9 represents an overview of the scenario.

Table 7.9.: Scenario 9: Portability

Aspects	Description
Causes	New platforms want to make use of SE-DSNL
Goals	SE-DSNL has been ported to a different platform
Quality Attributes	Performance and Stability can be affected by different implementations of the JRE
Approach	Either port the complete code or provide a web service which can be accessed from any platform
Risks	Eclipse or Java may not be available, which could make a port of the complete SE-DSNL framework difficult
Sensitivity Points	
Effort	Low to High

7.3.3. Performance

Assessing the performance of the SE-DSNL concept is difficult as it depends on many variables and algorithm. The core part which is responsible for the main computational time is the Construction application algorithm (as described in section 4.2.2). In the following, we only look at this part as the remaining algorithms (e.g., all implemented Functions or the SpreadingActivation) have a maximum worst case performance of $O(n^2)$. The same is true for the current implementation of the semantic information retrieval concept of chapter 6. Therefore, we analyze the worst case runtime performance of the Construction application process only. We make the following assumptions:

1. No optimizations or heuristics are applied.
2. None of the Constructions contain ConditionStatements, therefore, no Constructions can be filtered as all have the same result value. This further means that each Construction can be applied to any node in the syntax tree.
3. The SE-DSNL model contains a total of c Constructions.
4. All Constructions have the same amount of ConstructionSymbols s.
5. We assume that the syntax tree is a binary tree, i.e., each node has exactly two children.
6. Each leaf of the syntax tree has been mapped to the same amount of Construction instances m.
7. The syntax tree has a height of h at the root node and 1 at the leafs.

The runtime complexity is measured by the amounts of Construction instances which could be created in the scenario at the root of the syntax tree. As it was explained earlier, a bottom-up approach is used, in which for every node in the syntax tree all instances of all children are collected. This means that for each node at level 1 there are exactly m Construction instances, i.e., $f(1) = m$. For each node at level 2, there are $f(2) = (2^{2-1} * node_{lvl1})^s$. For nodes at level 3, there are $f(3) = (2^{3-2} * node_{lvl2} + 2^{3-1} * node_{lvl1})^s$ and so on. Generically this means that the number of Construction instances for a node of the syntax tree at level n (this must be a value within $[1..h]$) in the worst case scenario is equal to

$$f(n) := c * (\sum_{i=1}^{n-1} f(n-i) * 2^i)^s \tag{7.1}$$

and $f(1) := m$. As can be seen this is a recursive function which largely depends on the number of ConstructionSymbols s per Construction. For a tree of height 4 and $s := 2$, $c := 1$, $m := 4$ the result of $f(4) := 1,743,897,600$. The biggest problem is, therefore, the average number of ConstructionSymbols as this largely influences the number of potential new Construction instance. In tables 7.10 and 7.11 the difference can be seen immediately. For both calculations we assumed a tree of height $h := 3$. In the first table, only the number of available Constructions c has been changed, which leads to a big increase in Construction instances at the root node. However, a slight increase of s from two to three has a much bigger impact on the generated Construction instances. The reason is obvious: In the previous formula s is an exponent to the complete sum, whereas c only acts as factor of the product. However, as has been stated before, this is a worst case scenario and can not be computed. By using ConditionStatements and different heuristics the amount of instances could be restricted to a bare minimum in the prototypical implementation. This resulted in an average analysis time of 2.7s in case study 1 (section 7.4) which consisted of many more Constructions and ConstructionSymbols than the exemplary calculation.

Description	1 Construction	5 Constructions	10 Constructions
$h := 3, s := 2$	20736	2151680	16796160

Table 7.10.: Construction instances depending on varying numbers of Constructions

Description	2 Symbols	3 Symbols	4 Symbols
$h := 3, c := 1$	20736	1124864000	4,53889E+15

Table 7.11.: Construction instances depending on varying numbers of ConstructionSymbols

7.4. Case Study 1

The main focus of the thesis was to provide a concept as well as a prototypical implementation which allows to map language to the semantic information within a specific domain and parse natural language texts of the domain. Normally, any Domain Specific Language (DSL) has a limited set of elements and rules on how to build a structure between those elements. However, natural language itself is not limited. Even in a specific domain, the linguistic expressiveness must not necessarily be limited. An example is a company which provides online support and wants the incoming requests to be parsed automatically. Those requests do not necessarily apply to exact grammatical rules. This is problematic for the exact analysis of a textual request. Still, such domains have a need for mapping natural language texts to the available semantic information, too. Therefore, we want to look if it is possible to apply SE-DSNL in such a domain. In this case study we focus on the following questions:

1. How well can SE-DSNL perform in a domain with basically no restrictions regarding the linguistic expressiveness?
2. Can SE-DSNL take advantage of the combination of semantic and syntactic information?

The first question implies an open domain which is the opposite of what SE-DSNL was initially designed for. It is of course not possible to apply a closed domain concept to a completely open domain. However, we designed an ontology for a set of sentences and sentence pairs. Those were chosen randomly from a German online community, i.e., we created a linguistically closed domain, which however possesses a wide variety of syntactic structures. The second question relates to if the integration of semantic information into the analysis process can really be useful in the parsing process. The system is designed to check semantic as well as syntactic information in parallel as well as rely on available semantic information to generate relevant information (e.g., for WSD and pronominal anaphora resolution). Hence, some tests are run twice, i.e., the second time some of the Statements which relied on semantic information are disabled. We show how this affects the parsing results. All of this is shown in section 7.4.3. For this case study we decided to have a look at the car support domain. As input we selected some manually created as well as several randomly selected sentences and sentence pairs. The latter were randomly chosen from a German online car community (it should be noted that the parsing results of the manually created sentences do not differ much from the randomly selected texts). The sentences taken from the car community were checked for spelling errors and corrected. Based on those test sentences, an SE-DSNL model was developed. The ratio between manually created and online test sentences is about 1:2. Due to the randomly selected sentences, many ambiguities were introduced, both on a lexical as well

as a syntactical level. In the following, we describe how the case study was designed and what the results look like.

This section is structured as follows: Section 7.4.1 describes the overall design of the case study. It specifies the design of the SemanticScope, SyntacticScope as well as ConstructionScope. Next, we present the results of the evaluation in section 7.4.2, before discussing them in detail in section 7.4.3. The case study is concluded in section 7.4.4.

7.4.1. Case Study Design

As mentioned before the goal of the case study is to show how well the SE-DSNL concept can be used in linguistically complex domains. All semantically relevant information which is mentioned within the texts is contained within the SE-DSNL model. This means that all relevant lexical, syntactic and semantic information of every text exists within the SE-DSNL model. Relevant information is such which is represented by verbs, nouns as well as to a certain degree adjectives and adverbs. The semantic information has to be integrated within the generalization hierarchy as well as associated to related SemanticElements, such that the model as a whole conforms to the guidelines of section 3.5.4. Further as the semantic information should be constructed according to the meaning of the sentences, the potential syntactic structures must also be identified accordingly. All the information must be part of the SE-DSNL model.

We randomly selected a set of small text samples, which were written in a German online community about cars or handmade for test purposes. All test data can be seen in section A.2 of the appendix. Misspellings as well as major syntactical problems were corrected. Next, each of the test sentences was analyzed for its semantic information which was extracted and integrated into the SE-DSNL model. The process is described in section 7.4.1.1. Next, the semantic information was mapped to its corresponding Forms and phrases (specified in section 7.4.1.2). Finally all Constructions for the available syntactic structures have been created. This is shown in section 7.4.1.3.

7.4.1.1. Semantic Scope

As mentioned before, all the relevant factual semantic information was extracted from the sentences and put into an SE-DSNL model. This is shown in figure 7.3. It depicts the first sentence of text 16 in the appendix. Its English meaning would correspond to "I have to change the alternator of my Trajet, because it ran for 160000 km". We assume that not all the words in the sentence are of acute importance. Therefore, we focused on words which contain information that is more likely to have a high value to the semantic meaning of the complete text. Those words are "Ich" (English "I"), "bei" (English

Ich muss bei meinem Trajet die Lima wechseln, da sie jetzt 160000 drauf hat.

Figure 7.3.: Part of text 16 which shows the selected words which transport semantically relevant information

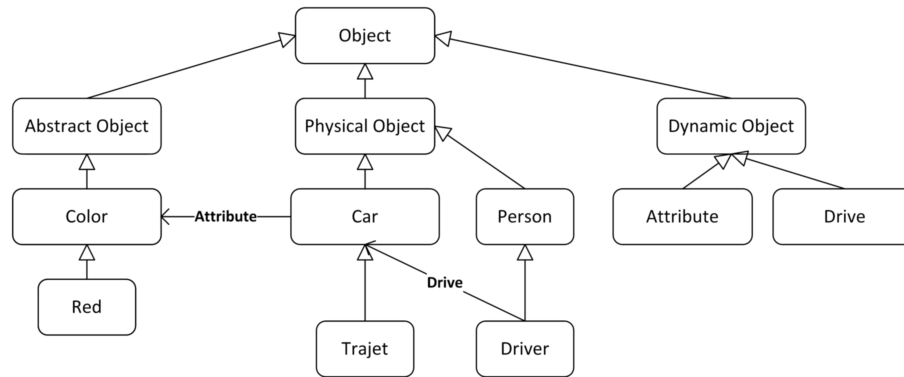


Figure 7.4.: Small excerpt of the SemanticScope which has been created for the first case study

"of"), "meinem" ("my"), "Trajet", "Lima" ("Alternator"), "wechseln" ("change"), "sie" ("it"), "160000" and "hat" ("has" / "ran"). Corresponding SemanticElements for all those words were created, i.e., the elements have been inserted into the Generalization hierarchy at a fitting position. Next, the SemanticElements have been connected with Associations. These structures represent information about things specific elements can do or be. A classic example is an Association of type 'Drive' between 'Driver' and 'Car'. Another example would be that a 'Car' has an Association of type 'Attribute' to the concept 'Color', indicating that a car can have a color. Figure 7.4 shows an excerpt of the SemanticScope containing the information. At the top, there is the element 'Object', which represents the root element of the SemanticScope Generalization hierarchy. There are three children (which we also call categories), named 'Abstract Object', 'Physical Object' and 'Dynamic Object'. The first one is a category representing all types of more abstract concepts like 'Color'. The second element 'Physical Object' represents, as the name implies, physical objects of all kinds, amongst which are 'Person' and 'Car'. The third element 'Dynamic Object' is the category which contains all those elements which are used for specifying the structural relations between the elements of the other two categories. Hence, the SemanticElement 'Attribute' is used to specify attribute-like relations between elements, e.g., from 'Car' to 'Color' and 'Drive' specifies one possible relation from 'Driver' to 'Car'.

One aspect which has already been mentioned in the previous parts of the thesis is that the SE-DSNL concept does not yet support a clear separation of instance and concept layer. This can also be seen in figure 7.4. Here, the element 'Trajet' could represent an

instance (as it is also being used in figure 7.3). However, there is no possibility to clearly distinguish between an instance element and a concept element. Therefore, the instance has been modeled as a normal SemanticElement which is a child of another SemanticElement. As a rule of thumb it can be said that only if an element is a leaf of the Generalization hierarchy it is possible that the element might be an instance. This means that 'Trajet' would be an instance in our context, however, e.g., the element 'Driver' is not, although it also is a leaf.

The final SemanticScope consists of 210 SemanticElements, 212 Generalizations and 50 Associations.

7.4.1.2. Syntactic Scope

The SyntacticScope is necessary for two reasons: First it provides all Forms which are used to represent SemanticElements, i.e., every element from the SemanticScope can be represented by single words. These are modeled as Forms or FormRoots within the SyntacticScope. An example is the representation for the SemanticElement 'Accelerate' whose German translation is "beschleunigen". Its word stem is "beschleunig", different possible inflections are "beschleunigt", "beschleunigen" and "beschleunige". Hence, we introduced a FormRoot, which represents the word stem "beschleunig". The FormRoot references the three previously mentioned different inflections. We did however not add all possible inflections to every form root but only those which were actually being used. Future applications could also be mapped to a database containing all different inflection forms and / or rely on similarity based string measurements in order to find the most likely form to match a specific word.

Some terms like numbers impose specific problems, i.e., there are infinite numbers and, therefore, also infinite possible representations. Therefore, we created a regular expression which matches strings that consist of digits only.

One aspect of the SyntacticScope besides containing all the different Forms is the representation of all possible SyntacticCategories which can be used by the syntactic parser to tag the words of a sentence as well as label the nodes within the syntax tree. For the case study we used the Stanford Parser 2.0.1 with its corresponding German parsing model. We of course need to be able to identify the different nodes of a syntax tree (for example the one seen in figure 7.6) correctly. Therefore, we put all the different POS tags and syntax tree node labels within one Generalization hierarchy. An excerpt of this can be seen in figure 7.5. The part which is shown there represents information about noun related categories. In order to create the structure, all categories were collected. Next they were classified to our best ability within a Generalization hierarchy. During experiments some-

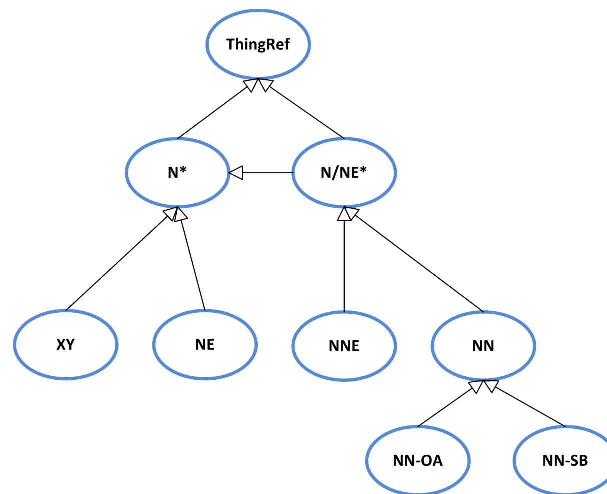


Figure 7.5.: Excerpt of the syntactic categories generalization hierarchy

times a situation A came up which showed similarities to another situation B that could already be parsed correctly. However, due to small differences between A and B, A could not be parsed with the same Construction as B. The differences between A and B were that the syntax tree for A used the syntactic category 'NE', whereas the one for B mentioned 'NN-SB'. As the Construction which had been successfully applied to situation B was using symbol 'NN-SB' directly, it could not yet be applied to situation A. In order to parse A and B with one Construction we first had to introduce a new SyntacticCategory which would be a parent to both 'NE' and 'NN-SB'. We called the category 'ThingRef', as shown in figure 7.5 (it does also reference other categories besides the previous two). Then the Construction was altered, i.e., the type of the SyntacticSymbol, which had been 'NN-SB', was changed to 'ThingRef'. From this moment on, the Construction was capable of parsing both situations A and B, as the referenced SyntacticCategory 'ThingRef' is a parent of both 'NE' and 'NN-SB'.

The process was applied in many situations, therefore, introducing new SyntacticCategories which were not directly part of the tag set of the syntax parser. However, the new information allows us to limit the amount of Constructions necessary to parse similar but not equal structures. This is done by introducing a certain degree of tolerance into designing Constructions by not using the most specific SyntacticCategories but more abstract ones. The process of relaxing precision as well as the ambiguities resulting from it is specified further in the following section.

The final SyntacticScope consists of 182 FormRoots (with mostly multiple different Forms), 121 SyntacticCategories and 133 Generalizations between the SyntacticCategories.

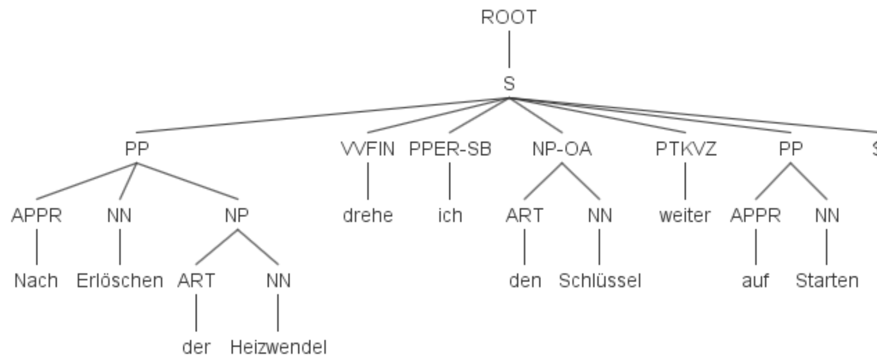


Figure 7.6.: Syntax tree of sentence 28

7.4.1.3. Construction Scope

The first created Constructions are the ones mapping the forms of the SyntacticScope to the SemanticElements. We do not explain the process, as it is actually relatively simple and has already been shown in section 3.5.2. In this section we focus on the process of how the Constructions have been created which map grammatical structures to the SemanticScope. We start by looking at the syntax tree in figure 7.6 and compare it to what the actual sentence says. The syntactic structure of the sentence matches the content of the sentence perfectly. The first part "Nach Erlöschen der Heizwendel" (English: "After the cool down of the heating coil") is correctly marked as a prepositional phrase. Next, the word "drehe" (English: "Turn") has been tagged as the verb of the sentence. The tag 'PPER' of the word "ich" (English: "I") has the small post fix 'SB', indicating that this seems to be the subject of the sentence. Next, the two words "den Schlüssel" (English: "the key") represent the noun phrase object of the sentence. "Weiter" (the best English translation would probably in combination with the word "turn" be "keep turning") is marked as a separated verb particle. The final two words "auf Starten" (English: "to start") are again a prepositional phrase. Here, the syntax parser delivered a good parsing result. Based on this we show how the Constructions have been built. We start at the beginning of the initial prepositional phrase. It is constructed out of four words: one preposition, two nouns and an article. Semantically the prepositional phrase is about the 'cool down' of the 'heating coil', where the 'heating coil' represents more specific information about the 'cool down' process. We, therefore, created a Construction which expects a syntactic structure as indicated by the prepositional phrase. The Construction can be seen in table 7.12. It obviously references four ConstructionSymbols 'cs1' to 'cs4', one for every word to be checked. Next, a SemanticSymbol 'obj' of the semantic type 'Object' has been created. It simply identifies if a word has a semantic meaning which is available within the ontology (i.e., the word has been mapped to the SemanticScope). Following, SyntacticSymbols for the different syntactic categories of the tree have been created. Note that 'N*'

Table 7.12.: Textual Representation Construction APPR + NN + ART + NN

Name	Content
Symbols	ConstructionSymbol Construction cs1 ConstructionSymbol Construction cs2 ConstructionSymbol Construction cs3 ConstructionSymbol Construction cs4 SemanticSymbol Object obj SyntacticSymbol Appr appr SyntacticSymbol ART art SyntacticSymbol N* n SyntacticSymbol PP pp
Condition Statements	inOrder(cs1, cs2, cs3, cs4) isOfType(cs1, appr) isOfType(cs2, n) isOfType(cs2, obj) isOfType(cs3, art) isOfType(cs4, n) isOfType(cs4, obj) isSyntacticallyRelated(cs1, cs2) isSyntacticallyRelated(cs2, cs3) isSyntacticallyRelated(cs3, cs4)
Effect Statements	addAttribute(cs2, cs4) representsSemanticSymbol(cs2) representsSyntacticSymbol(pp)

is the same element which can also be seen in figure 7.5. Next the different constraints are checked. Those involve the order of the words ('inOrder(cs1, cs2, cs3, cs4)') as well as the types of the single words. The first and the second noun are further required to have been mapped to the ontology. This is checked by the Statement 'isOfType(cs2, obj)' which validates if there is a SemanticElement 'Object' available within the referenced Construction instances. The mixture of syntactic and semantic type checks enables us to search for information of the correct type. However, those checks alone are not enough. We also have to specify that those words are not too far apart. This is done by checking the syntactic relations trough the Statement 'isSyntacticallyRelated'. It calculates how far the different words are apart from each other. The higher the return value of the Statement is, the closer the words are actually located together in the sentence. If this would not be validated the words could be dispersed over the whole sentence although belonging to completely different semantic entities. Introducing the syntactic distance bias does of course not directly prevent this from happening, however we assume that as we ideally have all required Constructions available, the ideal Constructions should also be the best match to its corresponding phrase structure. If all ConditionStatements have been

evaluated successfully, the EffectStatements will be executed. First of all, the semantic information of the instance behind the symbol 'cs4' is added to the semantic information of 'cs2' as an attribute. The result of the Statement is a generic AssociationInterpretation between 'cs2' and 'cs4', i.e., an AssociationInterpretation of type 'Object'. Further, the Construction instance which is the result of the Construction execution, represents the same SemanticElement as 'cs2' (indicated by 'representsSemanticSymbol(cs2)'). Further, it represents the SyntacticCategory 'PP'. A similar, although much simpler Construction has been created to correctly parse the prepositional phrase at the end of the sentence in figure 7.6.

In order to put the complete sentence together the Construction in table 7.13 has been created. The Construction is more complicated as the previous one. It especially incorporates direct checks of the information within the SemanticScope. This is shown in the following. The Construction consists of five ConstructionSymbols 'cs1' to 'cs5'. It further references three SemanticElements 'Object', 'Dynamic Object' and 'Process'. There further are three SyntacticCategories, i.e., 'VV*' (which matches anything that is a full verb), 'ThingRef' (which is a parent SyntacticCategory of both nouns as well as personal pronouns) and 'PP'. First the Construction checks for semantic information within the SemanticScope ('checkForTriple'). For this it only considers the information of those two ConstructionSymbols which are of type 'ThingRef', i.e., 'cs3' and 'cs4'. It requires that the SemanticElements behind the instances, which are matched to 'cs3' and 'cs4', are connected by an Association of the semantic type of the instance behind 'cs2', the verb of the sentence. The assumption is that if the information of the sentence is available within the SemanticScope this is an indication that the currently parsed structure is correct. However, the Statement can not yet exclude any given information. Instead it can only confirm if information exists. Next, the correct order of the words and phrases to be matched is validated. Note that 'cs2' is missing here. The reason is that the position of the verb often changes. In order to still apply the same Construction to similar situations, the correct order condition has been relaxed here. However, the verb is still required to be close to 'cs3', the probable subject of the sentence (as indicated by 'isSyntacticallyRelated(cs2, cs3)'). Additionally, 'cs1' should be syntactically close to 'cs3', 'cs3' to 'cs4' as well as 'cs4' to 'cs5'. Next, the different types of the instantiated ConstructionSymbols are checked. The instance of 'cs2' (the verb) is required to be of the semantic type 'Dynamic Object' instead of just 'Object', e.g., the concept 'Turn' for turning-a-key is a child of 'Dynamic Object'. As explained previously, the 'Dynamic Object' element subsumes all SemanticElements which are used to relate other elements. This introduces a higher degree of precision not only for the ConstructionSymbol 'cs2', but also for all remaining ConstructionSymbols. The reason is that in order to instantiate 'cs2' a more specific type of information must be used, which can not be used to instantiate any of the other ConstructionSymbols. Besides enhancing precision this also reduces computational complexity as fewer combinations

exist for how the different ConstructionSymbols can be instantiated. Aside from 'cs2', all other instantiated symbols are also required to contain semantic information. Besides the

Table 7.13.: Textual Representation Construction PP + V + ThingRef + ThingRef + PP

Name	Content
Symbols	ConstructionSymbol Construction cs1 ConstructionSymbol Construction cs2 ConstructionSymbol Construction cs3 ConstructionSymbol Construction cs4 ConstructionSymbol Construction cs5 SemanticSymbol Object obj SemanticSymbol Dynamic Object act SemanticSymbol Process proc SyntacticSymbol V* vv SyntacticSymbol ThingRef thing SyntacticSymbol PP pp
Condition Statements	checkForTriple(cs3, cs2, cs4) inOrder(cs1, cs3, cs4, cs5) isSyntacticallyRelated(cs1, cs3) isSyntacticallyRelated(cs2, cs3) isSyntacticallyRelated(cs3, cs4) isSyntacticallyRelated(cs4, cs5) isOfType(cs1, obj) isOfType(cs1, pp) isOfType(cs2, act) isOfType(cs2, vv) isOfType(cs3, obj) isOfType(cs3, thing) isOfType(cs4, obj) isOfType(cs4, thing) isOfType(cs5, obj) isOfType(cs5, pp)
Effect Statements	addAttribute(cs2, cs1) addAttribute(cs2, cs5) createTriple(cs3, cs2, cs4) findNewInfo(cs3, cs2, cs4) representsSemanticSymbol(proc)

ConditionStatements, the EffectStatements also have a higher degree of complexity. First the structure within the InterpretationModel is created. Hence, the semantic information is generically added to the semantic information of 'cs2', the verb (indicated by both 'addAttribute' Statements). Further, 'cs3' is connected to 'cs4' via the type of 'cs2'. Note the same order of arguments as in the ConditionStatement 'checkForTriple': If the ConditionStatements have been evaluated correctly, 'checkForTriple' will at least not have

failed or in contrast will have been evaluated successfully. Therefore, it is reasonable to connect the information in the same order as they have been validated in. Next, the Statement 'findNewInfo' is called with again the same arguments as before. The Statement adds completely new information to C^I (the set of all Construction instances, see definition 22), if it can find any, based on the given triple. The Statement, therefore, relies on the spreading activation algorithm (section 5). Finally the semantic type of the new instance, which is created as a result of the Construction application, is set to 'Process'. The element has been chosen because sentences often describe processes or situations.

The Construction shows precisely how both semantic and syntactic information are checked simultaneously during the Construction application process. All of the remaining Constructions have been created similarly. In total, there are 201 mapping Constructions (note that the number is higher than the amount of FormRoots available within the SyntacticScope, meaning that homonyms exist) and 12 Constructions for composite terms (i.e., mapping multiple words to one single SemanticElement). Further, a total of 63 grammatical Constructions have been created, similar to the two which were presented in this section.

7.4.2. Results

The results were gathered by parsing all sentences / sentence pairs based on the information available within the SE-DSNL model. The best rated InterpretationModel of every text was evaluated against what we expected to be the perfect result. In the following we present the results of the case study. The data can be seen in table 7.14. *The first row 'Total # texts'* describes the number of sentences / sentence pairs which have been analyzed in total. The test data can be seen in appendix section A.2). *The second row 'Completely correct results'* shows the number of sentences which have been parsed correctly, i.e., all relevant information are available within the InterpretationModel and all results contain useful and correct semantic types; Further the semantic structure within the InterpretationModel is correct and complete (i.e., no pronominal anaphora is missing). *The third row 'Completely correct results percent'* shows the percentage of correctly parsed results, i.e., all relevant information are available within the InterpretationModel, all results contain useful and correct semantic types; Further the semantic structure within the InterpretationModel is correct and complete (i.e., no pronominal anaphora is missing). *The fourth row 'Partially usable results'* shows the number of partially usable results, i.e., results whose semantic information is not as specific or accurate as they could be or which contain a wrong pronominal anaphora. Still, the semantic structure which has been created as a result of the Construction application must be correct. *The fifth row 'Partially usable results percent'* shows the percentage of partially usable results, i.e., results whose seman-

ID	Description	Value
1.	Total # texts:	32
2.	Completely correct results:	26
3.	Completely correct results percent:	81.3%
4.	Partially usable results:	3
5.	Partially usable results percent:	9.35%
6.	Not usable results:	3
7.	Not usable results percent:	9.35%
8.	Texts with solvable pronominal anaphoras:	10
9.	Correctly resolved pronominal anaphoras:	8
10.	Falsely resolved pronominal anaphoras:	5
11.	Texts with more specific information:	8
12.	More specific information correctly identified:	7
13.	Average parsing time:	2.7s
14.	Minimum parsing time:	0.8s
15.	Maximum parsing time:	21s

Table 7.14.: Results of the first case study

tic information is not as specific or accurate as they could be or which contain a wrong pronominal anaphora. Still, the semantic structure which has been created as a result of the Construction application must be correct. *The sixth row 'Not usable results'* specifies the number of unusable results, i.e., sentences from which not all semantic information was used or for which wrong Constructions have been applied, leading to incorrect semantic structures. *The seventh row 'Not usable results percent'* specifies the percentage of unusable results, i.e., sentences from which not all semantic information was used or for which wrong Constructions have been applied, leading to incorrect semantic structures. *The eighth row 'Texts with solvable pronominal anaphoras'* describes sentences which contain solvable pronominal anaphoras, i.e., pronouns which a precedent noun. *The ninth row 'Correctly resolved pronominal anaphoras'* states the number of sentences from row eight whose pronominal anaphoras have been correctly resolved. *The tenth row 'Falsely resolved pronominal anaphoras'* specifies the number of sentences in which pronominal anaphoras have been resolved mistakenly. The number refers to the amount of total sentences tested, not the amount of text with solvable pronominal anaphoras. *The eleventh row 'Texts with more specific information'* defines the number of sentences in which the semantic information could be made more specific based on the information within the ontology. *The twelfth row 'More specific information correctly identified'* specifies in how many of the sentences of the previous row the more specific information has been found correctly. *The 13th, 14th and 15th row* represent the average, minimum and maximum time in seconds that the complete parsing process for one sentence took (starting with the syntax parser, applying the Constructions and extracting the InterpretationModels) on an Intel Core i5-2400 system with 8GB RAM, Windows 7 x64 and JDK 6U27; The Stanford parser used

was version 2.0.1 with its corresponding German PCFG model.

7.4.3. Discussion

The previous section shows an overall good result of 81.3% correctly parsed texts although the tested scenario is not optimally suited for SE-DSNL. The number means that out of the 32 sentences, 26 were parsed correctly, including correctly disambiguated semantic information, correctly identified more specific information and pronominal anaphora resolution. In the following, we explain especially the partially as well as not usable results and what lead to them. We start with the unusable results.

As was explained while describing the case study design in section 7.4.1 we tried to extract and add all relevant information from the test texts to the SE-DSNL. Still not every text was correctly parsed. The first two sentences which yielded wrong results were 15 and 28:

- 15: Ich muß bei meinem Trajet die Lima wechseln, da sie jetzt 160000 drauf hat. Außerdem lädt sie unter 2500 UPM nur manchmal. (English: I have to change the alternator of my Trajet, because it has run for 160000. Further, she only sometimes works correctly below 2500 UPM)
- 28: Es dauert ungefähr 20 Sekunden, bis der Anlasser den Motor startet (English: It takes about 20 seconds until the starter starts the engine)

We begin with the second sentence 28. It is a relatively simple sentence (in contrast to others from the test set). Still the best rated InterpretationModel is not the correct one. A Construction has been applied to the first part of the sentence ("Es dauert ungefähr 20 Sekunden") which is not the ideal match. It means that a wrong structure has been created out of the first part of the sentence until the comma. The actual problem here is that the correct InterpretationModel is available but has not been rated the highest. The reason is the following: A Construction has been applied to the first part which is more simplistic (i.e., it contains fewer Statements and references fewer ConstructionSymbols) than the ideal Construction for this part. More complex Constructions tend to get lower likeliness values when being evaluated. The reason can be seen in definition 21 and how the value v_{st} is calculated: It is the product of all return values of the executed Statements. As soon as there is one Statement with a very low return value or multiple Statements with lower return values, the whole v_{st} value drops accordingly. Normally, this is not a problem as the v_{int} introduces a bias based on how much of a sentence a Construction actually covers. However, in the best rated InterpretationModel the wrong Construction does not cover the complete first part of the sentence, i.e., it only covers the sentence starting at "dauert", whereas in the correct InterpretationModel the correct Construction

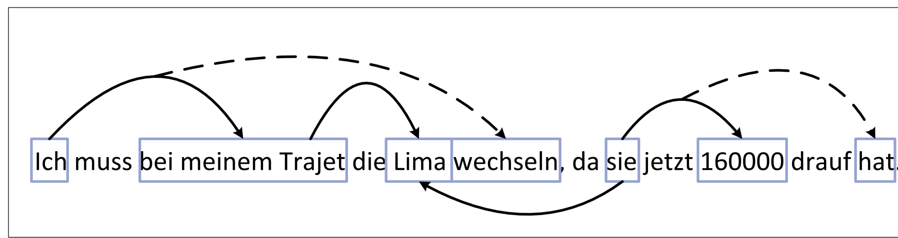


Figure 7.7.: Part of text 16 which shows the selected words and the structure the Constructions creates

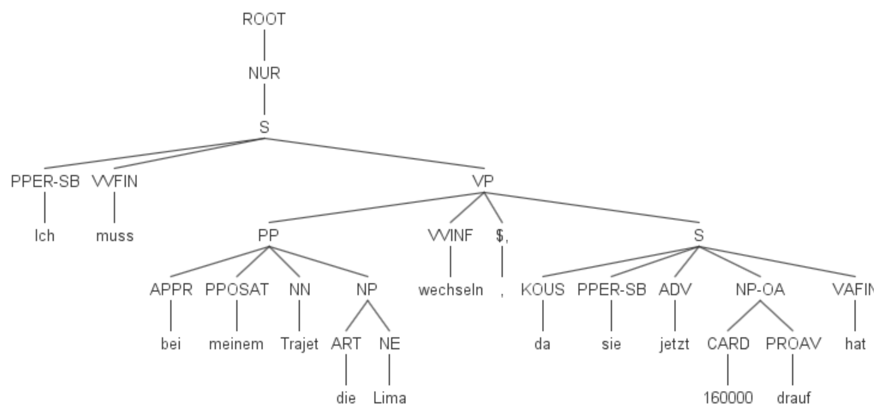


Figure 7.8.: Syntax tree of the first part of text 15

starts at "Es". Still, the difference in textual coverage is not sufficient to rate the correct InterpretationModel higher than the actually best rated one. Hence, the correct result is available in the final solution set, however it has not been rated accordingly.

The second unusable result is the parsing result for text 15. In section 7.4.1.2 we have used a part of text 15, which can be seen in figure 7.3. The simplified interpretation of the parsing process is seen in figure 7.7 (the straight lines indicate an Association, the dashed lines going out from the straight lines point to the type of the Association). Ideally, "Ich" should reference "Lima", the type of the Association being "wechseln". However, "Ich" references the semantic information of "bei meinem Trajet", which is connected to "Lima" with a generic Association. The reason lies within the syntax tree, which is shown in figure 7.8. We focus on the first prepositional phrase of the sentence. The problem with the phrase structure is that it is not optimal. Normally, only "bei meinem Trajet" should be a 'PP' whereas 'die Lima' should be a noun phrase which is not subordinated to the prepositional phrase. This leads to a wrong Construction being applied, to be more precise the Construction which we have shown in table 7.12. The Construction is especially intended for structures like the ones in figure 7.6, but not this one here. It is also impossible to differentiate the 'PP' structures of both sentences, i.e., there is no additional syntactic information available which could be helpful in refining the Construction such

that it would only be applicable to the sentence of figure 7.6. Even on a semantic level it is difficult to imagine what kind of information must be available to correctly identify the structure. The problem is that in both sentences the meanings behind the nouns or named entities are also semantically related (as each 'Car' has an 'Alternator' and a 'Heating coil' can 'cool down'). Perhaps it could be helpful to check for an 'Attribute' or 'Ability'-like relation between the existing concepts, e.g., an 'Alternator' has the 'Ability' to 'Cool Down'. In contrast, there is no such relation between 'Car' and 'Alternator'. However, it can be thought of other situations which contain the same 'PP' structure but don't have an 'Attribute' or 'Ability' like Association between the referenced SemanticElements. As can be seen, the situation is difficult to tackle, especially because of an incorrect syntax tree.

Another problem with text 15 happens in its second sentence "Außerdem lädt sie unter 2500 UPM nur manchmal." The problem here is the pronoun "sie", which references the word "Lima" from the first sentence. The pronominal anaphora has not been resolved correctly, as the syntactic distance from the pronoun in the second sentence to the word "Lima" in the first sentence is too big. Another problem occurs with pronoun "sie" in the second part of the first sentence. Because of the wrong 'PP' structure the pronominal anaphora does not reference the word "Lima" but instead references the information of "meinem Trajet". This clearly shows how much a small variation in the syntax tree can still affect the overall result of the prototypical SE-DSNL implementation, although the syntax tree is only used as a soft alignment structure.

The next sentence with unusable results is sentence 39: "Meine Motorkopfdichtung ist defekt, ich muss immer Kühlflüssigkeit nachfüllen." (English: "My engine head gasket is broken, I always have to refill coolant"). The sentence shows two problems:

1. The word "ich" is marked by the syntax parser as a personal pronoun, therefore, our pronominal anaphora algorithm tries to resolve it. However, there is no corresponding noun available in the sentence. Still, the algorithm identifies a possible anchor and inserts the corresponding information. This is a classic example of a false positive.

2. The other problem is that in this case there is no Construction available which can fully parse the second part of the sentence. This means that either the SemanticElements of "ich", "immer" and "nachfüllen" are put together or "ich", "Kühlflüssigkeit" and "nachfüllen". The sentence would partially be usable, if the best rated InterpretationModel would at least reference the Construction which uses the latter three words. However, the Construction is again more complicated as the Construction for the best rated InterpretationModel. Therefore, the wrong result is again rated better than the correct result, as it has already been the case in the previous text 28.

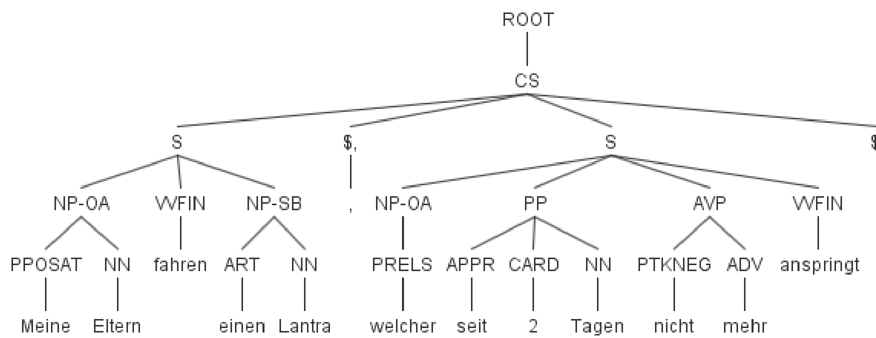


Figure 7.9.: Syntax tree of sentence 17

This could probably be fixed with a new Construction for specifically the syntactic structure. However, is has to be checked if this might lead to new ambiguities.

Regarding the partially usable results we begin with sentence 17: "Meine Eltern fahren einen Lantra, welcher seit 2 Tagen nicht mehr anspringt." (English: "My parents drive a Lantra which doesn't startup since 2 days"). The corresponding syntax tree is available in figure 7.9. The reason the sentence was rated as only being partially correct, is that the pronominal anaphora of "welcher" has not been resolved correctly. Instead of referencing the SemanticElement of "Lantra" it references the part of "Meine Eltern". There are two reasons for this behaviour:

1. The verb of the second part of the sentence (after the comma) has been modelled as a compound phrase, i.e., "anspringt" and "nicht" have been mapped together on a single SemanticElement 'NotStartup'. The problem is that the information is only available at a later stage in the analysis process, i.e., only when the process reaches the second 'S' node. The reason is that the analysis process is based on a bottom-up approach. At this stage in the process, the pronominal anaphora resolution has already been applied to the word "welcher", however with a wrong verb (i.e., the SemanticElement of the single word "anspringt", which is exactly the opposite of "nicht anspringt").

2. Even if the pronominal anaphora resolution algorithm would have received the correct arguments, it would have been doubtful that it could have identified the correct information. The problem is that no information has been created within the SemanticScope which allows to identify that a car may not startup, i.e., a corresponding Association is not available.

Therefore, the reason for sentence 17 not being parsed correctly is not exactly a shortcoming of the SE-DSNL concept, but a not existing feature in the prototypical implementation as well as missing information in the SemanticScope.

The second partially usable sentence is the one with ID 27 "Nach Erlöschen der Heizwendel drehe ich den Schlüssel weiter auf Starten." (English: "After cool down of the heat coil I keep turning the key to start"). The sentence also has a false pronominal anaphora because of the word "ich", which has no preceding noun that actually represents a SemanticElement of type 'Person'. This is also a classical false positive.

The last partially usable sentence 29 also shows the same problem, i.e., a false positive pronominal anaphora: "Wenn ich nicht länger als eine halbe Stunde warte, dann startet er ganz normal." (English: "If I do not wait for more than an hour, it starts up normally"). Here, the problem lies within the pronoun "er", which actually refers to a car, however there is no such noun preceding the pronoun. Still, the pronominal anaphora algorithm detects a seemingly correct mapping, therefore, a pronominal anaphora is introduced into the final result.

In the previous part of the section we elaborated the negative results of this case study and why they occurred. One thing that is left is showing that the integration of semantic information in the analysis process is possible and helpful. Basically there are three ways how the semantic information is incorporated into the parsing process of the SE-DSNL prototype.

1. Word Sense Disambiguation: By continuously checking if information is available within the SemanticScope, our approach directly integrates those results into the parsing process. To better validate the Statement we changed a single parameter of the prototype, which controlled the string similarity threshold in order to match a word from a sentence to a Form within the SyntacticScope. It has the effect that a word is more likely to be mapped to multiple different Forms. Based on this we reevaluated sentence 1: "Nachdem die Person im Auto sitzt, kann sie das Auto starten und fahren" (English: "After the person sits in the car, she can startup and drive the car"). Due to the changed parameter, there were now four homonyms: "Nachdem" (mapped to two different SemanticElements), "sitzt" (mapped to three SemanticElements), "starten" (mapped to three SemanticElements) and "fahren" (mapped to three SemanticElements). This means a total of $2 * 3 * 3 * 3 = 54$ possibilities. However, all of the elements except the word "starten" were correctly disambiguated because of the semantic checks.

2. Pronominal anaphora resolution: As we presented previously in the results table 7.14, 8 out of 10 possible pronominal anaphoras were correctly identified. As shown in section 4.3.2, the pronominal anaphora algorithm relies to a large degree on the available semantic information. Hence, without the semantic information, the precision rapidly drops as the only other remaining feature (the syntactic distance) is not enough to correctly identify the right anchors.

3. More specific information: People often tend to talk about things in a very generic way (see the section 2.1.4 about vagueness). Specifying the mentioned information is easy for humans but difficult to do for computers. Our approach is capable of extracting the information from the SemanticScope, if the information is available in specific situations. This can be seen, e.g., in sentence 12: "Wenn ich die Warnblinkanlage einschalte, dann höre ich ein Geräusch" (English: "If I turn on the warning indicator, I hear some noise"). The word "ich" is mapped to the SemanticElement "Person" only. However, in the InterpretationModel, the SemanticElement 'Person' has been replaced with 'Driver', which is correct, based on the given context. To compute the result, the algorithm from section 5 received the parameters 'Person', 'Turn On' and 'Warning Indicator'. It found a corresponding connection between 'Driver', 'Turn On' and 'Warning Indicator'. This had the effect that a new instance of type 'Driver' for the word 'ich' was introduced. The complete analysis process was repeated with the new information, which in the end was rated higher than the similar result with the less specific information 'Person'.

These results clearly show that the direct integration of semantic information is possible and useful. All three areas for which semantic information have been used, showed good results despite their prototypical nature.

7.4.4. Conclusion

The first case study showed that it is possible to use the SE-DSNL concept to create a prototypical implementation which is capable of parsing even a wide variety of linguistic expressions. It further showed that incorporating semantic information from ontologies can be useful in solving different parsing tasks like WSD, pronominal anaphora resolution or enhancing the precision of the semantic information.

However there are of course certain restrictions within this case study. The size of the test data is not comparable to real world data. Larger data sets would certainly introduce more problems, one of them surely being performance. As the case study has shown, performance is not yet suited for problems of similar dimensions. It would, therefore, require further optimizations. This would especially be true for larger data sets. Also, the creation of Constructions has proven to be difficult and time consuming because of ambiguities, some of which have also been shown in the previous chapters. Therefore, finding a good balance between parsing precision and tolerance as well as performance is difficult (the more tolerant Constructions are, the worse the performance of the overall system becomes because more Constructions can be applied to one and the same context and thereby increasing the amount of instances at runtime, which again leads to more possibilities for creating new Construction instances etc.).

Aside from those more general challenges, the problems which have been found during this case study, can be classified into one of the following four categories:

1. Problematic pronominal anaphora: If the sentence contains a resolvable pronoun (i.e., the noun which the pronoun refers to, is available in the preceding part of the sentence) and the information is available within the SemanticScope, there is a high probability for the pronominal anaphora to be resolved correctly (80% precision). However, if one or both are missing, many false positives are detected (overall five). Changes to the way the pronominal anaphora algorithm works are therefore necessary to solve the problem.

2. Not fully implemented features: Some problems came up due to not yet implemented features. One is that pronouns are not yet being rechecked for possible anchors if a compound verb has been identified at a later stage of the parsing process.

3. Not optimally rated Constructions: Finding a good balance on how the likeliness of Construction instances is being rated is difficult. The current values have proven to yield good results, but these could even be optimized. Especially the mixture of the Statement return value and interval ratio is difficult to weight, as has been shown in the discussion section.

4. Missing information: Although we started the case study with the assumption that all relevant information should be contained within the ontology, not all have been added. This can be problematic for matching all available syntactic and lexical information correctly. Further, it might lower the precision in identifying the correct word senses or resolving pronominal anaphoras properly. In real world scenarios the problem is most probably even more pressing as real world scenarios require a constant knowledge refinement and update cycle.

Either of the first three categories can be fixed or optimized in the future, thereby increasing the overall precision of the system. The last point however is more difficult to treat. There are, however, domains in which a limited amount of information is enough to correctly apply the SE-DSNL prototype. We show such an example in the second case study.

7.5. Case Study 2

The first case study showed that SE-DSNL can be used in domains which do not impose specific restrictions on syntactic structures. However, such domains currently come with several disadvantages, especially regarding the runtime performance of the analysis, which may last several seconds on long and grammatically complicated sentences. Also, the design of a set of Constructions can be very time consuming.

In our second case study we therefore focused on a scenario which is better suited for the original intentions of SE-DSNL, i.e., parsing texts of a specific kind in a specific domain. A topic which has gained a lot of attention in the last couple of years is controlling software by using natural language, i.e., the user can either speak or type a natural language command which is analyzed and executed by an application. The probably most famous example at this time is Siri[2] from Apple. It allows a user to state commands which are interpreted and (in case of a successful interpretation) executed by Siri accordingly. Commands can be such things like searching for information, ordering a seat at a restaurant or more humorous things like asking Siri for a joke.

This area is very interesting for SE-DSNL as it shows many of the characteristics which SE-DSNL was developed for:

1. The domain is limited by the requirements of the application, i.e., the domain contains a finite set of things which can be done. Further, the changes within the domain are better measurable and can, therefore, be integrated within an ontology more simply. The user can only control what is available in this restricted space.
2. If a domain should be controllable with natural language, it is common to restrict the linguistic expressiveness of the user. This makes it easier for a software component to parse the information a user expresses, which improves the precision of the analysis and parsing process. This is due to less possibilities of how the user can state his / her natural language commands. Further, the fewer information there is, the lower the probability of ambiguities becomes.

Both facts ideally match the prerequisites of SE-DSNL. We, therefore, demonstrate in this case study that SE-DSNL can be used to make an application controllable by using natural language commands. The goals of the case study are that

1. a controlled natural language can be parsed with high precision in SE-DSNL,
2. the commands can be classified by using the Pattern concept (see chapter 6) and
3. the arguments which are necessary to execute a command can be extracted correctly

[2]http://www.apple.com/de/ios/siri/

from the command.

The section is structured as follows: In section 7.5.1 first the overall design of the case study is explained. Following, it describes how the ontology, its Constructions as well as the Pattern have been designed. Next, the results of the case study are presented in section 7.5.2, before we discuss the results in 7.5.3. The case study is concluded in section 7.5.4.

7.5.1. Case Study Design

For this case study we decided to design a fictitious project management software. The software should allow users to do anything that supports them in managing their projects and teams. The user has the possibilities to add employees to teams, initiate calls, send an email or write a new task. Further, the user should be able to retrieve different information, e.g., mails about specific projects or from specific employees. A full list of all commands which have been used for the evaluation, is available in the appendix section A.3.

The case study is designed as follows: We first explain in section 7.5.1.1 what type and kind of information the SemanticScope must contain. Following in section 7.5.1.2, we explain how we mapped the information of the SemanticScope to the elements of the SyntacticScope. Next we describe the simplified syntactic structures available in our controller natural language (section 7.5.1.3) and the Constructions we developed in order to parse them. The section also explains how the semantic information of a command are connected within an InterpretationModel. Finally, section 7.5.1.4 explains the process of designing Patterns which are able to correctly classify commands.

7.5.1.1. Semantic Scope

As we described briefly in section 3.5 gathering the semantic information can be done either automatically (if the information is available within an OWL ontology) or manually. For the case study we had to develop the information within the SemanticScope using a manual approach. A lot of the semantic information that is needed for controlling a software with natural language is available in the meta model of the application's data structure and the instances of the meta model. For example, the meta model of a project management software does most likely contain the information that, e.g., employees are part of project teams, employees have more specific child classes like project members and project leaders etc. This is shown in figure 7.10. The presented model is already in line with our guidelines of section 3.5.4.

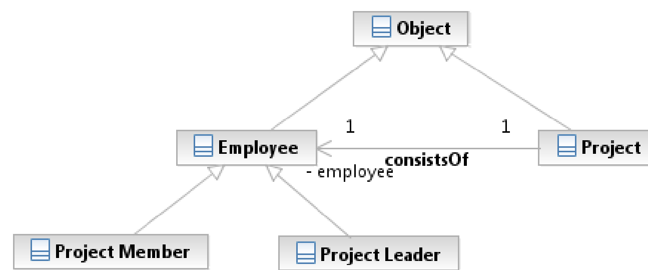


Figure 7.10.: Excerpt of a UML metamodel for a project management software

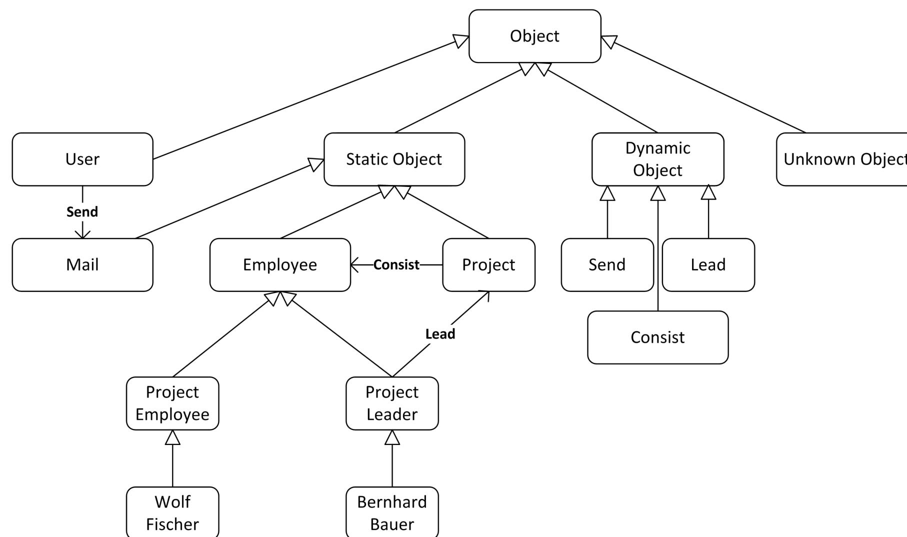


Figure 7.11.: Simplified excerpt of the Semantic Scope for case study 2

The information of the meta model is both useful as well as usable for our project, we, therefore, insert the information directly into our SemanticScope, part of which can be seen in figure 7.11. The 'Object' element of the previous meta model has been replaced by the element 'Static Object' in the SemanticScope. Further, we introduced new elements for the types of the Associations between the UML classes, e.g., a new element 'Send' which is used as the actual type of the relation between 'Project' and 'Employee'.

Overall, the SemanticScope has been differentiated into four different branches, two of which are 'static' objects (i.e., physical objects, projects, etc.) and the other ones for dynamic objects and concepts, i.e., elements which normally describe relations between static objects (this is in compliance with guideline 5 which requires that the types of Associations are part of a different Generalization branch). Besides the already mentioned relation between 'Project' and 'Employee' we specified that a 'Project Leader' obviously 'leads' a 'Project'. Further, we do not separate the instance level from the concept level, therefore, instances are directly contained within the SemanticScope (e.g., the elements 'Wolf Fischer' and 'Bernhard Bauer').

The two other Generalization branches below the root object are 'User' and 'Unknown Object'. The 'User' has been modeled as an element which is completely independent of the remaining objects within the ontology. The reason is that the program in our case does never know exactly who is currently using the program. Thus, in order to have a clear and clean separation between the representation of the user in the ontology and all the remaining elements it was decided to make him a distinct element.

The 'Unknown Object' is an element which is used to 'map' unknown words to it, i.e., words which have no mapping to other SemanticElements from the 'User', 'Static Object' or 'Dynamic Object' Generalization paths. Normally, in a situation with an unknown word, the semantic spreading activation algorithm could be of help by 'guessing' the best suitable concept in a specific context. However, in this case study, we assume that unknown words are most of the times new names for already existing elements (e.g., if the user wants to rename something) or new names for new elements (e.g., if the user wants to create a new task or project). This means that commands may contain unknown words intentionally as the command creates or renames an existing element. In order to handle those cases we decided to use a default element which all unknown words are mapped to. This helps increasing the precision as unknown elements can easily be spotted. Thereby, elements can easily be created or renamed.

The final Semantic Scope consists of a total of 99 SemanticElements, 25 Associations and 99 Generalizations.

7.5.1.2. Syntactic Scope

The SyntacticScope contains all the SyntacticCategories which are required to map the results of the syntax parser and the SE-DSNL model. The SyntacticCategories are similar to those of the first case study, as we again use the Stanford Parser for creating both the POS tagging as well as the syntax tree. The main difference is that the commands are given in English. However, the labels of the tree nodes are basically the same as in the first case study (see section 7.4.1.2). Besides the SyntacticCategories, the SyntacticScope contains all lexical information for the SemanticElements of the SemanticScope.

A special case poses the element 'Unknown Object' as explained earlier. To linguistically represent it, we created a regular expression which basically matches to any string and maps it to the element 'Unknown Object'. However, the regular expression would map every word additionally to the 'Unknown Object', leading to a huge increase in computational complexity (as additional elements would have to be considered during the Construction application process). Therefore, we made a small change to our implementation such that only those words which have not been mapped to another SemanticEle-

ment besides 'Unknown Object', are mapped to 'Unknown Object'.

7.5.1.3. Construction Scope

We first created all mapping Constructions from Forms to their corresponding SemanticElements, thereby closing the gap between the syntactical representation and the SemanticScope. We continued with creating the complex Constructions. Natural language commands for a software can be expressed in many different forms. In order to limit complexity we created a simple controlled natural language. Commands should, therefore, come in one of the following shapes (C refers to the grammatical structure of the command itself, V to a Verb, NP to a noun or noun phrase and PP to a prepositional phrase; for the remaining tags have a look at section A.1):

1. $C := V + NP + PP$: A verb is followed by a noun and a prepositional phrase. Example: 'Add Enrique to Development'
2. $C := V + NP + NP$: A verb is followed by two nouns. Example: 'Show me my mails'
3. $C := V + NP + NP + PP$: A verb is followed by two nouns and a prepositional phrase. Example: 'Give me the phone number of Enrique'

A syntax tree can contain certain substructures which themselves represent a NP or a PP. Those structures are: $NP := NP + PP \mid NP + NP \mid DT + NN \mid PRON + NN \mid DT + JJ \mid DT + JJ + NN \mid PRP + JJ + NN$ and $PP := APPR + NP$. All of the commands in appendix section A.3 apply to one of the three structures above. Accordingly we created three Constructions for the three grammatical structures. All three Constructions were designed the same way: They check the correct order and types of the single words / phrases as well as relate them accordingly in the InterpretationModel. We decided to relate the semantic meaning of the words in the same order as the words are related within the sentences, i.e., in a sequential order. Table 7.15 represents the Construction which can parse the first command structure from above. It enforces the correct order of all referenced ConstructionSymbols 'cs1', 'cs2' and 'cs3'. Following, the different types of the ConstructionSymbols are checked. We decided to use a mixture of precision and tolerance in checking types, i.e., if we check for syntactic features, there is a relatively high precision involved. This means, ConstructionSymbol 'cs1' must be of SyntacticCategory type 'VB' (this category only has a single child within the SyntacticCategory Generalization branch which means that it is a very specific element). 'cs2' must be of the syntactic type 'PP', which is a leaf of the SyntacticCategory tree. In contrast, we have a very tolerant approach to checking the semantic constraints. We simply require each of the three ConstructionSymbols to be mapped to one SemanticElement. This helps us in parsing words / phrases which are known within the SemanticScope, i.e., there is a Construction mapping the correspond-

Table 7.15.: Textual Representation of Construction V + NP + PP

Name	Content
Symbols	ConstructionSymbol Construction cs1 ConstructionSymbol Construction cs2 ConstructionSymbol Construction cs3 SemanticSymbol Object obj SemanticSymbol Command cc SyntacticSymbol PP pp SyntacticSymbol VB vb
Condition Statements	inOrder(cs1, cs2, cs3) isOfType(cs1, obj) isOfType(cs1, vb) isOfType(cs2, obj) isOfType(cs3, obj) isOfType(cs3, pp) checkUserAction(cs1, cs2) isSyntacticallyRelated(cs1, cs2) isSyntacticallyRelated(cs2, cs3)
Effect Statements	addAttribute(cs1, cs2) addAttribute(cs2, cs3) representsSemanticSymbol(cc)

ing word to a SemanticElement. Note that we do not check the second symbol 'cs2' for a syntactic type. The reason is that the syntax parser sometimes created syntax trees in which the second argument was classified as an adjective instead of a noun (this again brings us to our previous statement about the problem with parsing precision in section 4.2). Hence, we decided to relax the constraints in this aspect. Still, we could hold up a high precision regarding the final result, as is shown later.

After checking the different types of the Constructions, we next introduce a ConditionStatement which is specific to this case study. It validates if the user is capable of executing the action mentioned in this specific command. This is done by calling the Function 'checkUserAction' which receives 'cs1' and 'cs2' as input parameters, i.e., the element 'User' within the SemanticScope should have an Association of the type 'cs1' to the SemanticElement of 'cs2'. For example, if 'cs1' represents the SemanticElement 'Send' and 'cs2' represents 'Mail', then 'checkUserAction' checks if there is an Association from 'User' to 'Mail' of the type 'Send'. For the ontology in figure 7.11, the Statement, therefore, returns 1.

After 'checkUserAction', the ConditionStatements 'isSyntacticallyRelated' introduce a bias based on the syntactic distance between the phrases / words behind the three ConstructionSymbols. The smaller the distance is between the three phrases / words, the

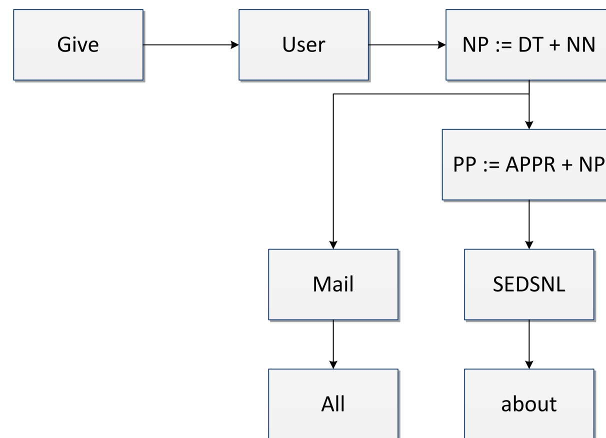


Figure 7.12.: Simplified InterpretationModel for the command 'Give me all mails about SDSNL', representing only the ConstructionInterpretation elements

better.

If the ConditionStatements can be executed successfully and the return values are good enough, the EffectStatements are executed. The 'addAttribute' Statements add the information of the Constructions behind 'cs2' as an attribute to the information of 'cs1'. This is done generically by relating the two corresponding InterpretationElements with an AssociationInterpretation of the semantic type 'Object'. As 'cs1' is related to 'cs2' and 'cs2' is related to 'cs3', a sequential order has been created.

Finally, 'representsSemanticSymbol' is called which states that the newly created Construction instance is of the semantic type 'Command' (the symbol 'cc' references the SemanticElement 'Command').
Note that although the Statement creates a sequential semantic structure, this does not imply that every InterpretationModel simply consists of a sequential semantic structure. The reason is that noun phrases can contain sequential semantics which together with other noun phrases form tree like structures. An example can be seen in figure 7.12. It shows a simplified InterpretationModel of a command with the structure $C := V + NP + NP + PP$. At the beginning of the InterpretationModel, there is the Construction interpretation element representing the action 'Give', followed by the first argument 'User' and a noun phrase ConstructionInterpretation, which represents the SemanticElement 'Mail'. The noun phrase is followed by the InterpretationElement for the prepositional phrase, which itself is connected to the 'SEDSNL' element. As can be seen, the model has a tree like structure with the branching starting in the node $NP := DT + NN$.

We built a total of eleven complex syntactic Constructions which are sufficient to parse all tested commands (note that this is not even $\frac{1}{5}$th of the Construction amount of the

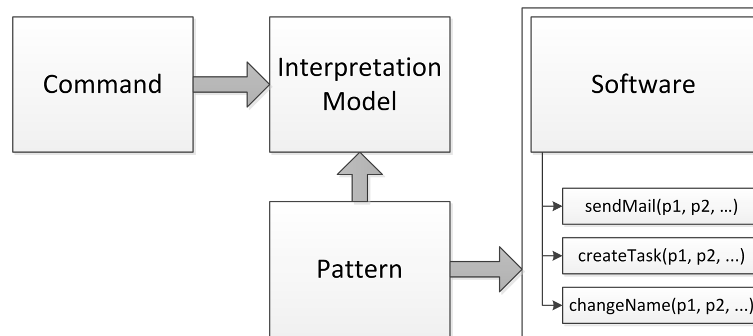


Figure 7.13.: Overview of the Pattern mechanism in the second case study

first case study, as seen in section 7.4.1.3: Syntactic complexity and potential ambiguities has been decreased by a large amount). Further, five Constructions for mapping composite terms were created (i.e., Constructions which map compound terms to a single SemanticElement, e.g., a Construction for the compound term "project leader").

7.5.1.4. Pattern Scope

Patterns allow to classify InterpretationModels and retrieve information from them. This is based on available semantic information and the structure between SemanticElements (for more details have a look at chapter 6). In this case study we need Patterns to classify the commands accordingly to the functions which are available within our fictitious project management application. An overview of the architecture is available in figure 7.13. It answers the basic question, how information within a given command can actually be used to control a software application. In order to use the information of the command we need to know which of the words within in the command apply to which parameter of which function of the software. An example would be the command 'Send mail to Wolf', where the word 'Wolf' would be the parameter for a function 'sendMail(receiver)'. However, all the information within the command is needed to classify the command and associate it to the correct function. If for example the command is 'Read mails from Wolf', it would require a completely different software function. Also, a similar sounding command like 'Send message to Wolf' could actually mean a different function like 'sendSMS(receiver)', depending on what the semantic information of the word 'message' is. We show in the following how the Patterns for the case study have been designed in order to differentiate similar commands.

We began by classifying the commands according to the functions which the software application should provide. We came up with a total of 18 different patterns that are required to classify the commands as well as map the parameters correctly. Next we created each of the 18 patterns. A Pattern consists of basically three things:

1. PatternElements of a specific semantic type, which represent the semantic information that should or must be available within the InterpretationModel
2. PatternRelationships between the PatternElements, which specify the structure that should exist within the InterpretationModel
3. Attributes like transitivity and necessity that further specify or relax certain conditions

We continued with selecting the elements that are required for satisfying a Pattern. As an example we have a look at commands 25 and 26 from the appendix section A.3:

25. Give me all emails from Enrique
26. Give me the last email from Enrique

We assume that both sentences (as well as all others from the same category) can be handled by the same software function, i.e., a function which returns one or several mails which correspond to certain criteria specified in the command. To start the Pattern creation process we first focused on the common information within both commands. Both mention the action 'Give', both contain the concept 'Mail' as well as a person by the name 'Enrique'. Therefore, it is reasonable to assume that a Pattern which should detect both commands must contain elements of the semantic types 'Give', 'User', 'CommunicationElement' (which is the parent element of the SemanticElement 'Mail') and 'Employee' (which is the parent element of the SemanticElement 'Enrique'). These four build the core of our new Pattern as seen in figure 7.14 (the representation within the figures is not a 100% accurate as normally each PatternElement would have a reference on its corresponding SemanticElement; Because of clarity, the representation means that the name of the PatternElement matches the one of its SemanticElement and, therefore, each shown PatternElement also implicitly references a corresponding SemanticElement). However, the figure shows two other elements 'Order' and 'Amount'. In command 25 there exists the word 'all'. It indicates a certain *amount* of mails which should be returned to the user. Command 26 in contrast mentions the word 'last', which implies a specific email from an ordered list. Both words transport information which is relevant to executing the commands such that the user is satisfied. We, therefore, added the elements 'Order' and 'Amount' to the Pattern. However, we have to treat them distinctly from the rest as we show later.

For now, all the required semantic information of the Pattern have been gathered. Yet missing is the structure. As we explained previously in section 7.5.1.3 we simply connect the information within the InterpretationModel in a sequential order with generic relations (i.e., the type of the relations is always of type 'Object'). As we know what the structure of the InterpretationModels looks like we connect the elements of the Pat-

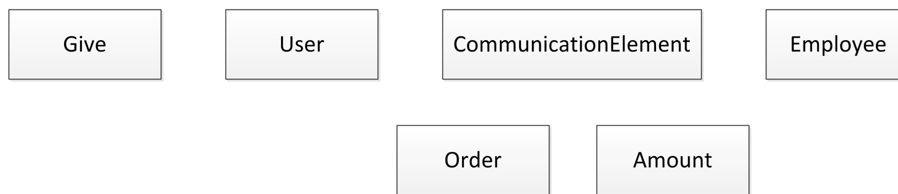


Figure 7.14.: First step of creating a Pattern is selecting the correct elements

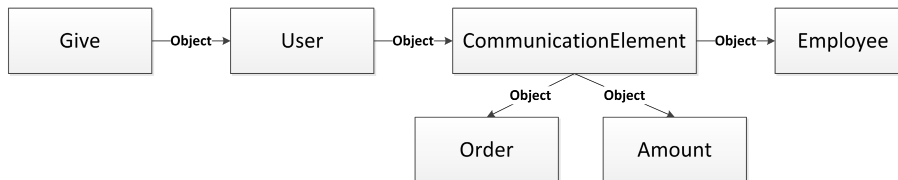


Figure 7.15.: Second step of creating a Pattern is adding the relations which represent the required structure within the InterpretationModel

tern similarly. This can be seen in figure 7.15. In a sequential way 'Give' is connected to 'CommunicationElement', which itself is related to 'Employee'. Also, 'CommunicationElement' is connected to both 'Order' and 'Amount'. At this point, the Pattern could only be solved on an InterpretationModel, which represents the very same structure as the Pattern at this point. This of course is a problem due to two different reasons:

1. Neither command 25 nor 26 contain all the information of the current Pattern. More precisely, command 25 only contains information about the amount of information, but not about the position of the email in a specific order. In contrast, command 26 only contains information about a specific position of an email in a given order, but not about any amount.

2. As seen in figure 7.12, the structure of the InterpretationModel does not necessarily represent the most cohesive structure possible. Instead, the information can be a bit distributed because there may be InterpretationElements which represent grammatical Constructions only. Those elements simply indicate which InterpretationElements have been put together using which Construction. An example is the element $NP := DT + NN$ as seen in figure 7.12.

We, therefore, have to relax the constraints of the Pattern such that it can be applied correctly to the InterpretationModels of both commands. This is done as follows:

1. Necessity: Both the 'Amount' and the 'Order' are not present within both commands. Hence, we cannot treat them as core elements of the Pattern, although they contain relevant information. This means that we want to be able to extract the information from the commands if it is available. However, it should not hinder the Pattern from being applied correctly if the corresponding information is not

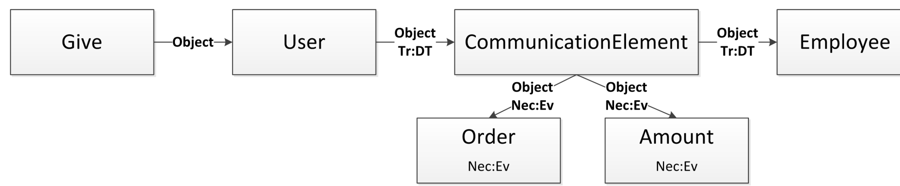


Figure 7.16.: Third step of creating a Pattern is setting the required attributes

available. Therefore, instead of requiring both elements as well as their incoming relations to be always available, we change their necessity attribute values to 'Eventually', i.e., on the relations from 'CommunicationElement' to 'Amount' as well as from 'CommunicationElement' to 'Order' and on the PatternElements 'Amount' and 'Order'. The change allows the Pattern to be resolved correctly, even if neither or just one of both elements is available within the InterpretationModel.

2. Transitivity: As shown before, the structure of the InterpretationModels may differ from that of the Pattern. However, it is not so much the structure itself as the number of InterpretationRelations between specific elements (as can be seen in figure 7.12 in which actually two steps exist between the element 'User' and the element 'Mail'). Hence, we have to relax the transitivity attributes of the PatternRelationships from 'User' to 'CommunicationElement' as well as from 'CommunicationElement' to 'Employee' by changing their values to 'Directed Transitivity'.

The result of the changed attributes can be seen in 7.16. 'Tr:' represents the 'Transitivity' attribute which has been set to 'DT' ('Directed Transitivity'). Further, 'Nec:' is the shortcut of the attribute 'Necessity' whose value has been set to 'Eventually' ('Ev' is its abbreviation). With these attribute changes the Pattern now matches both commands mentioned above as well as all other commands of the same category (for a complete list, look at section A.3 in the appendix). All other 17 Patterns were built in a similar way.

7.5.2. Results

All commands (see the complete list in section A.3 of the appendix) were matched against all available Patterns. The results can be seen in table 7.16. *The first row 'Total # commands'* represents the amount of commands tested; *The second row 'Total # Patterns'* specifies the amount of Patterns available within the SE-DSNL model; *The third row 'Correctly resolved commands'* defines, for how many of the commands the highest rated of the correctly solved Patterns was the intended Pattern (according to the previously created manual classification); *The fourth row 'Correctly mapped commands'* describes how many of the correctly resolved commands in row three were mapped correctly to its Pattern, i.e., which information of the command were mapped to its intended PatternElement; *The fifth row*

'Commands with multiple Patterns instantiated' shows the number of commands, which were matched to more than one Pattern based on the SemanticElements only (i.e., ignoring the structural information within the Pattern; This is the result of the first phase in section 6.4.1); *The sixth row 'Commands with more than one matching Pattern'* means the number of commands for which more than one Pattern has been solved correctly, including the structural information of the Pattern (i.e., the result of phase two in section 6.4.2) *The seventh, eighth and ninth row* represent the average, minimum and maximum time in milliseconds that the complete parsing process took for one command (starting with the syntax parser, applying the Constructions, extracting the InterpretationModels and finally applying the Patterns) on an Intel Core i5-2400 system with 8GB RAM, Windows 7 x64 and JDK 6U27; The utilized Stanford parser was version 2.0.1 with its corresponding English PCFG caseless model.

ID	Description	Value
1.	Total # commands:	35
2.	Total # Pattern:	18
3.	Correctly resolved commands:	34
4.	Correctly mapped commands:	34
5.	Commands with multiple Patterns instantiated:	7
6.	Commands with more than one matching Pattern:	1
7.	Average parsing time:	147ms
8.	Minimum parsing time:	46ms
9.	Maximum parsing time:	486ms

Table 7.16.: Results of the second case study

7.5.3. Discussion

The most important information gathered from the results is that it is possible with SE-DSNL to achieve a very high precision in parsing commands, if the available grammatical structures of the language have been limited. In this case, 34 out of 35 commands have been parsed correctly which corresponds to a precision of 97%. The average time that a complete parsing process took was 147ms. Those numbers suggest that the SE-DSNL concept is in fact suited for the task. In the following, we discuss the numbers and circumstances that lead to them in greater detail.

As can be seen, 7 of the 35 commands have initially been matched to more than one Pattern, i.e., the first phase of the Pattern evaluation process as described in section 6.4.1 has been finished successfully for more than one Pattern for some commands. Those commands as well as their Patterns can be seen in table 7.17. The first table column describes the ID of the command, the next column the Patterns which have been matched successfully after the first phase (as described in section 6.4.1), the column 'Phase 2' the

Patterns which remained after the second phase (as shown in section 6.4.1), the column 'Best Rated' the Pattern which was rated the highest and finally column 'Intended' the Pattern which was actually intended for the command.

Comparing the number of Patterns which could be matched to the commands at those different phases is interesting as the result of the first phase can be compared to an enhanced keyword based matching process: More simplistic approaches to classifying texts often rely on identifying all the keywords relevant to a specific category. A similar mechanism is performed in the first phase of the Pattern evaluation process, i.e., potential Patterns are selected based on the identified SemanticElements within the InterpretationModels of the corresponding commands. As this is a semantic instead of a syntactic level many problems like synonyms and homonyms have already been dissolved at this stage which of course is an advantage over normal keyword based approaches. However, ambiguities still remain as seven commands have been matched to more than one Pattern. To eliminate these ambiguities the structural information within the Pattern is valuable, which can be seen in the column 'Phase 2'. For six out of those seven commands the structural check helps identifying the correct (i.e., intended) Pattern.

Command ID	Phase 1	Phase 2	Best Rated	Intended
8	50, 52	50	50	50
10	50, 52	50	50	50
11	50, 51, 52	51	51	51
21	100, 130	130	130	130
28	170, 190	170, 190	190	170
29	100, 170	170	170	170
32	170, 180	170	170	170

Table 7.17.: Commands which have been matched to more than one Pattern in the first phase of the Pattern evaluation

The advantages of having a structural matching is shown for command 11. A simplified InterpretationModel of the command is available in figure 7.17. Note that the InterpretationModel representation does not contain any grammar related ConstructionInterpretation, i.e., these would normally be located where the wave-symbols are positioned. Therefore, e.g., between 'User' and 'Mail' is not just one InterpretationRelation but several as the ConstructionInterpretations are missing in the figure. The sequential order of the ConstructionInterpretations is apparent. The two prepositions 'In' and 'Out' lead to a tree like structure as they branch out of the 'trunk'. The question is, why the three Patterns 50, 51 and 52 can be matched to command 11 in phase 1 of the Pattern evaluation.

First we look at Pattern 50 in figure 7.18. The Pattern consists of four elements: 'Create', 'Note', 'OrganizationalUnit' and 'NoteName'. The first element 'Create' is obviously a

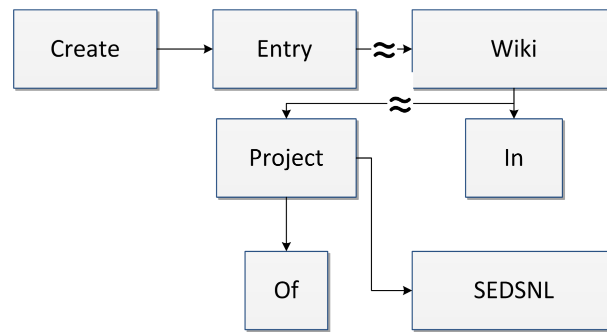


Figure 7.17.: Simplified InterpretationModel for command 11: 'Create a note in the wiki of project SEDSNL'

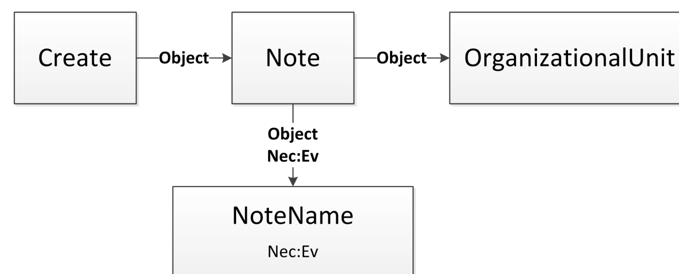


Figure 7.18.: Structure of Pattern 50

part of the InterpretationModel. The SemanticElement 'Note' is a parent of the element 'Entry' in the command. The semantic type of the PatternElement 'OrganizationalUnit' is a parent of the SemanticElement 'Project'. The PatternElement 'NoteName' is not required to be part of a corresponding InterpretationModel, therefore, it does not have to be matched. As all three required PatternElements can be mapped to the Interpretation-Model, phase 1 of the Pattern evaluation can be completed successfully. However, the Pattern can not be solved correctly in the structural evaluation. Although the first relation between 'Create' and 'Note' is found (between 'Create' and 'Entry'), resolving the PatternRelation between 'Note' and 'OrganizationalUnit' is not possible. The reason is that between the ConstructionInterpretations 'Entry' and 'Project' there is not one but multiple relations. This would, therefore, require a transitive relation, however, the according attribute has not been set. Therefore, the Pattern fails to be resolved correctly.

Next, we describe the evaluation of Pattern 51 (figure 7.19) on command 11. The mapping is similar to the one of Pattern 50. 'Location' has been defined as a parent of 'Wiki' in the SemanticScope, therefore, these two elements can be mapped. We won't go into detail about the remaining mappings as they are obvious. Note, however, that all elements have to be mapped as all necessity attributes have their default value, i.e., 'Always'. For the structural matching nearly all relations have been set to 'Directed Transitivity'. This way

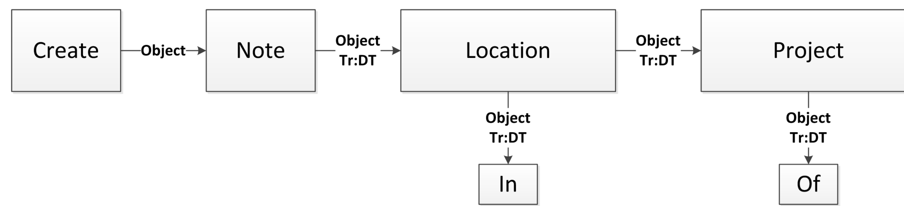


Figure 7.19.: Structure of Pattern 51

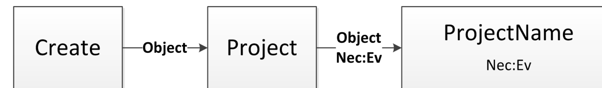


Figure 7.20.: Structure of Pattern 52

the relations between 'Entry' and 'Wiki' as well as 'Wiki' and 'Project' can be resolved correctly.

The last Pattern 52 is shown in figure 7.20. It consists of only three elements. The first two elements can be matched easily to the InterpretationModel. The last element, however, can not and does not have to be matched as its necessity attribute is set to 'Eventually'. In the second Pattern evaluation phase, the only remaining relation between 'Create' and 'Project' can not be solved correctly as there are too many steps between the 'Create' and 'Project' elements within the InterpretationModel. This shows that having a structural match increases the overall precision.

In contrast to the previous examples, command 28 has not been classified as intended. However, we show in the following that the result is not immediately wrong. Figure 7.21 represents its simplified InterpretationModel. The structure is also tree like with the branching beginning after the InterpretationElement 'Mail'. Pattern 170 is actually very similar to the Pattern which we created in section 7.5.1.4. There are some small differences which can be seen in figure 7.22: The element 'Employee' as well as the relation from 'CommunicationElement' to 'Employee' both have an 'Eventually' necessity. The task of Pattern 170 is to classify commands, which are about reading, e.g., all mails from a specific employee. In contrast, Pattern 190 (which is shown in figure 7.23) should return all mails about a specific topic. Hence, it had to be modeled in a way which allowed to capture any topic a mail could be about. The problem is that the concept 'Topic' is difficult to grasp. Theoretically this could be anything, e.g., a project, a specific employee, a company etc. We decided to simply ask for an 'Object' (as this must be the parent element of all other elements in the ontology). However, we require the 'Object' element to be related to the preposition 'About' such that this is not too arbitrary.

If one compares both Patterns, it becomes obvious that both are nearly the same. Both contain the elements 'Give', 'User', 'CommunicationElement' and 'Amount' (even with

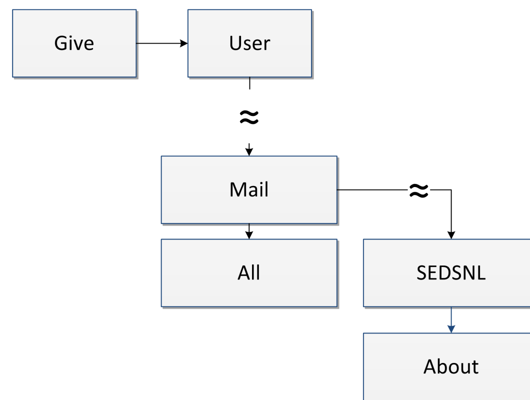


Figure 7.21.: Simplified InterpretationModel for command 28: 'Give me all mails about SEDSNL'

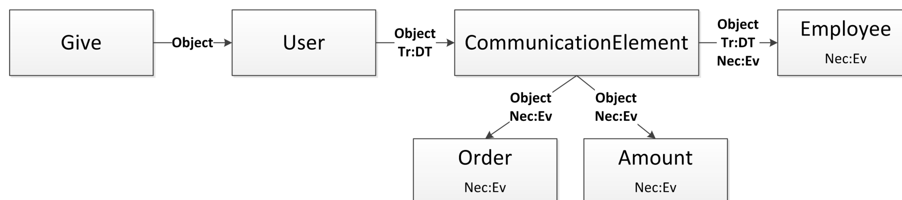


Figure 7.22.: Simplified InterpretationModel for Pattern 170

the same necessity attribute). The only difference is at the end, where Pattern 170 has an 'Employee' element only. In contrast, Pattern 190 has the 'Object' element with the addendum 'About'. However, the 'Employee' element in Pattern 170 is not mandatory. Therefore, Pattern 170 can in certain contexts be seen as a smaller and less specific version of Pattern 190. Command 28 represents exactly such a context which allows both Patterns to be solved correctly. Hence, this is not actually a wrong result, it is just not as intended due to an imprecise Pattern. The problem could be solved by, e.g., defining PatternElements which are explicitly *not* allowed to exist in an InterpretationModel. This means that specifying the absence of the 'About' element in Pattern 170 would solve the problem.

We finally discuss the performance numbers shown in the result section. They indicate that it is possible to create a system based on SE-DSNL that runs fast enough for a real

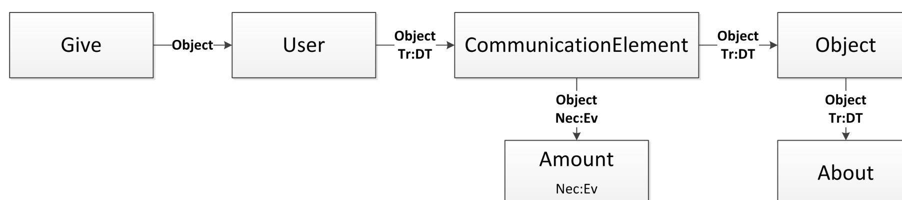


Figure 7.23.: Simplified InterpretationModel for Pattern 190

world application, i.e., there is no notable delay for the user. In the scenario it is not so much the linguistic complexity which limits the performance but the number of Patterns which are available within the model. However, the prototypical implementation has not yet been optimized for the best possible performance. Therefore, if the amount of Patterns would be increased by a larger factor, it is certainly possible to optimize the implementation and still retain a good performance.

7.5.4. Conclusion

In this case study SE-DSNL as well as the Pattern based information retrieval and classification concept was used to parse commands, classify them according to the functions of a project management software and further retrieve the required input parameters for the software functions. The results as well as the discussion showed conclusively that the requirements from section 7.5 can be fulfilled, even with a prototypical implementation. A small but efficient set of Constructions was used to parse the natural language command and create the InterpretationModels. Next, a set of Patterns was applied to the interpretations.

As the high precision indicates, both the Construction application process as well as the semantic information retrieval and classification work well. This fulfills requirement 2, i.e., by using the Pattern concept the commands could be classified correctly. This further fulfills requirement 1 (i.e., a controlled natural language can be parsed with high precision in SE-DSNL): If the parsing process would deliver 'unusable' results, the Patterns could not have been designed in a common, simple way and still deliver such a high precision. Also, as our manual validation showed, if the Pattern classified a command properly, the commands information was always mapped correctly to the corresponding PatternElements, therefore fulfilling requirement 3 (i.e., by using the Pattern concept the arguments which are necessary in order to execute a command can be extracted correctly from the command itself).

From these results we conclude that SE-DSNL can successfully be applied to domain specific real world problems.

8

Conclusion

8.1. Summary

The initial motivation behind SE-DSNL was to develop a system which bridges the gap between semantic and linguistic knowledge. We collected several requirements which we thought would be necessary for such a system. In the following paragraphs, a summary of the works which have been done in this thesis is given. We will further show if and how the different parts of SE-DSNL concept fulfill the initially specified objectives.

1. Combining Ontologies with Natural Language: In section 3 we introduced the meta model which is at the center of SE-DSNL. At its core it consists of three different scopes: One for ontological information, one for lexical and syntactic information and a third one which creates a bridge between the first two scopes. The model allows the integration of external Functions which makes it possible to tailor the SE-DSNL framework to any domain. This means that both arbitrary ontological as well as linguistic information can be represented. Further we showed how knowledge from existing OWL ontologies can be transformed to SE-DSNL and to which guidelines such knowledge must conform. The validity of the model has been demonstrated in both case studies in chapter 7. Therefore, the objective from section 1.3.1 is fulfilled.

2. Concurrent Analysis, Semantic Interpretation and Linguistic Phenomena: After the definition of the meta model, we specified the algorithm to concurrently analyze a natural language text and create an InterpretationModel from it (see section 4.2). It requires a SE-DSNL model whose Constructions are concurrently applied to a natural language text. The process is aligned to a syntax tree. The Statements of each Construction are executed and checked. Only if all ConditionStatements of a Construction return a successful value, the Construction is instantiated and its EffectStatements can be executed. Further we specified a set of different Statements which, based on the semantic information within the SE-DSNL model, solve linguistic tasks like pronominal anaphora resolution, word-sense disambiguation, vagueness and reference transfer. For the latter three, a spreading activation based approach was developed (chapter 5) which, based on a given triple input, discovers elements which are more likely to conform to the given input. The process together with the mechanisms of the different Statements has successfully been evaluated as shown in chapter 7.

   These concepts fulfill several of the initially stated objectives. First, the requirement "Concurrent Analysis of NLP specific problems" (section 1.3.2) is fulfilled, as the overall process is based on single Constructions which are put together concurrently, i.e., the result of one Construction can be used by another Construction,

which in turn creates new information which may be used by another one and so on. Next, the result of the Construction application process is an InterpretationModel, i.e., a model which represents the semantic interpretation of the natural language input. This fulfills the requirement "Semantic Interpretation of Natural Language Text" (section 1.3.3). The final requirement is "Handling Ambiguities, Vagueness and Reference Transfer" (section 1.3.4), i.e., handling problems like WSD and vagueness with the available semantic knowledge. These problems are solved by the spreading activation based algorithm from section 5.

3. Semantic Information Retrieval: In chapter 6 a concept was introduced which allows the retrieval of information from InterpretationModels and their classification as well. The process is based on Patterns, which specify how a set of SemanticElements should be related to each other. Patterns can also be refined by specifying certain attributes like transitivity. Next, a graph-matching algorithm tries to resolve the previously defined Patterns on a given InterpretationModel. The concept fulfills the requirement "Semantic Information Retrieval" from section 1.3.5, which has been shown in the second case study (section 7.5).

It can be concluded that all of the initially specified objectives have successfully been fulfilled by the SE-DSNL approach and its prototypical implementation.

In contrast to existing OBIR and OBIE systems, SE-DSNL introduces many different aspects which have not or only partially been considered by state-of-the-art approaches. The concept for mapping lexical, syntactic and semantic information is powerful and can be used with arbitrary languages which is a clear advantage over most of the existing systems. Other recent approaches like LexInfo also show that this is an important challenge and must be further elaborated.
The concurrent analysis has rarely been looked at. NLP in general and OBIE as well as OBIR often rely on a pipelined processes. In the past decade, some approaches used joint inference in OBIE systems [221] [222], however most approaches still rely on pipelined concepts (as shown in section 4.4). SE-DSNL is the first approach to our knowledge which combines the advantages of a flexible and adaptable concept (by using Construction) with the advantages of a concurrent approach (i.e., mutual disambiguation).
One final aspect which this thesis contributes, is a first idea and prototypical realization of how underspecification and reference transfer can be handled in a computational domain. Both are phenomena which are rarely treated by available components. Only a few concepts try to solve vagueness (e.g., Prince and Sabah [223]), however we know of no computational approach which handles reference transfer. The SE-DSNL concept shows how these challenges (besides WSD) can be solved in specific situations using only one algorithm and the available domain knowledge.

Many questions have come up during the development which are worth to be considered for future research. Some of these have been presented in section 8.2 and are already being worked on. The currently prioritized task is the feasibility evaluation of integrating SE-DSNL in Model-driven development (MDD) approaches, which shows promising results and will lead to a prototypical implementation.

8.2. Outlook

The concept and framework that have been developed in this thesis have only a prototypical status. There are many ways of improving the current state. We describe some of these potential tasks in the following.

1. Reducing the effort of Construction creation: Constructions are currently created manually, i.e., one expert creates the grammar for a specific language. Further each Construction has to know how to semantically enrich the interpretation (as this is the core result of the analysis process). However, a manual process is very time consuming and error-prone. Therefore, it is an important task to reduce the effort required to create a Construction. One approach would be to automatically deduce this information from a given database. The problem is that there are no sources which present a cohesive semantic representation of sample sentences, i.e., a semantic alternative to treebanks is missing. Current semantic evaluations mostly focus on senses of single words only. Another approach could be crowdsourcing [224] [225], i.e., outsourcing the task to a group of people each of which handles a small portion of the challenge. Creating Constructions could be ideally suited for crowdsourcing. Either, each expert would be given a single sentence or text which has to be parsed by SE-DSNL or he / she would be given the task of developing one specific Construction. This divide and conquer approach could speed up the Construction creation tremendously. However, aspects like security and trust have to be considered.

2. Machine-Learning for Construction application: Machine-learning approaches are known to show better results as their application is more flexible to different situations. The current concept already has the ability to assign probabilities to Constructions. That implies that given the context a Construction does appear in and the number of times this happens, it could receive a probability value. However, as a corresponding training set for semantic information is missing, it has not yet been developed and implemented. Still it could be evaluated how probabilities deduced from a mainly syntactic database would affect the overall performance of the analysis step.

3. Enhancing precision: Several of the developed algorithms, e.g., for WSD or pronominal anaphora resolution, yield good results in identifying the correct information if all required information is available within the SE-DSNL model. However, in case that certain information is missing or the problem is not solvable, the precision suffers, resulting in false positives. This was especially shown in the first case study, where pronominal anaphoras were falsely resolved. Hence, the algorithms computing the corresponding tasks should be improved to increase the

overall precision and performance.

4. Enhancing runtime performance: The evaluation showed that in smaller domains the SE-DSNL framework can be applied successfully, both in terms of precision as well as computational performance. However, the performance drops if the domain knowledge increases. One goal, therefore, should be to optimize the application algorithm. Well known in language processing are Finite-State Transducers (FSTs) [226] [20] for which efficient implementations are available. The challenge is to transform the Constructions with their varying Statements into a FST. However, the prospect of a better runtime performance might be worth the effort.

5. Adding new information to the SemanticScope: Currently the SE-DSNL framework creates a semantic interpretation which contains the mappings of which words correspond to which SemanticElements and which syntactic structures express which semantic information. The InterpretationModel can however also contain new information, both single elements as well as new relations between those elements. As part of an IE component it would be helpful if new information would automatically be added to the existing SemanticScope.

6. Natural Language text production: So far, the algorithms of SE-DSNL have been implemented to only parse natural language text. However, the underlying concept could also be used to produce natural language text, i.e., starting with the knowledge within the SemanticScope specific information would be selected and expressed with natural language. Such a process would require some new Statements and algorithms, however certain aspects could also be reused. Similar work has been done by Steels and Beule in FCG [46]. The production of text from knowledge can be helpful in a variety of scenarios, e.g., if the system needs feedback about specific knowledge, it can ask the user to validate it. The advantage is that the user would not need to manually search for the corresponding knowledge structures, but would receive the information in a form which he / she is used to.

7. Integration into Model-driven development: MDD describes concepts which generate code from models. An example is the Graphical Modeling Framework (GMF) [227]. It requires several models and can automatically create a fully-working application from them. Due to the technological similarity between the implementation of SE-DSNL and GMF it should be possible to integrate SE-DSNL directly into the MDD process, i.e., the user would have to create some additional SE-DSNL specific models during the development process. After the application is generated, it would immediately contain a component which allows the software to be controlled with natural language commands (similar to the ones from the second case study in section 7.5). The feasibility of this idea is currently being evaluated.

A

Appendix

A.1. POS Tags and Syntactic Categories from Penn Treebank

1.	CC	Coordinating conjunction	25.	TO	to
2.	CD	Cardinal number	26.	UH	Interjection
3.	DT	Determiner	27.	VB	Verb, base form
4.	EX	Existential there	28.	VBD	Verb, past tense
5.	FW	Foreign word	29.	VBG	Verb, gerund / present participle
6.	IN	Preposition	30.	VBN	Verb, past participle
7.	JJ	Adjective	31.	VBP	Verb, non-3rd ps. sing. present
8.	JJR	Adjective, comparative	32.	VBZ	Verb, 3rd ps. sing. present
9.	JJS	Adjective, superlative	33.	WDT	wh-determiner
10.	LS	List item marker	34.	WP	wh-pronoun
11.	MD	Modal	35.	WP	possessive wh-pronoun
12.	NN	Noun, singular or mass	36.	WRB	wh-adverb
13.	NNS	Noun, plural	37.	#	Pound sign
14.	NNP	Proper noun, singular	38.	$	Dollar sign
15.	NNPS	Proper noun, plural	39.	.	Sentence-final punctuation
16.	PDT	Predeterminer	40.	,	Comma
17.	POS	Possessive ending	41.	:	Colon, semi-colon
18.	PRP	Personal pronouns	42.	(	Left bracket character
19.	PP	Possessive pronouns	43.	)	Right bracket character
20.	RB	Adverb	44.	"	Straight double quote
21.	RBR	Adverb, comparative	45.	‘	Left open single quote
22.	RBS	Adverb, superlative	46.	“	Left open double quote
23.	RP	Particle	47.	’	Right open single quote
24.	SYM	Symbol	48.	”	Right open double quote

Table A.1.: POS tagset of the Penn Treebank [228]

1.	ADJP	Adjective phrase
2.	ADVP	Adverb phrase
3.	NP	Noun phrase
4.	PP	Prepositional phrase
5.	S	Simple declarative clause
6.	SBAR	Clause introduced by subordinating conjunction
7.	SBARQ	Direct question introduced by wh-word or wh-phrase
8.	SINV	Declarative sentence with subject-aux inversion
9.	SQ	Subconstituent of SBARQ excluding wh-word or wh-phrase
10.	VP	Verb phrase
11.	WHADVP	Wh-adverb phrase
12.	WHNP	Wh-noun phrase
13.	WHPP	Wh-prepositional phrase
14.	X	Constituent of unknown or uncertain category

Table A.2.: Syntactic tagset of the Penn Treebank

A.2. Case Study 1 Texts

ID	Text
1	Nachdem die Person im Auto sitzt, kann sie das Auto starten und fahren.
2	Eine Person hört Musik. Sie sitzt im Auto.
3	Beim Beschleunigen geht der Motor aus.
4	Wenn ich von der Kupplung gehe, geht der Motor aus.
5	Wenn ich von der Kupplung gehe, geht der Motor aus. Er stottert erst noch.
6	Wenn ich von der Kupplung meines Polos gehe, geht der Motor aus.
7	Der Motor geht aus, nachdem ich von der Kupplung meines Polos gehe.
8	Mein Auto verliert an Leistung, wenn es kalt ist.
9	Mein Auto stottert, nachdem ich es starte.
10	Wenn das Auto kalt ist, geht der Motor nicht an.
11	Mein Blinker leuchtet nur 1 mal, und der Warnblinker leuchtet auch nur manchmal.
12	Wenn ich die Warnblinkanlage einschalte, dann höre ich ein Geräusch.
13	Ich habe mir jetzt einen Mini USB Stick von Zinc Classic Line mit 4GB gekauft. Er funktioniert nicht im Auto.
14	Ich besitze einen neuen I20 ohne Handbuch.
15	Ich muß bei meinem Trajet die Lima wechseln, da sie jetzt 160000 drauf hat. Außerdem lädt sie unter 2500 UPM nur manchmal.
16	Wie kann ich den Keilrippenriemen entspannen?
17	Meine Eltern fahren einen Lantra, welcher seit 2 Tagen nicht mehr anspringt.
18	Meine Motorkopfdichtung ist defekt, ich muss immer Kühlflüssigkeit nachfüllen.
19	Beim Beschleunigen machen sich im Motorraum Geräusche bemerkbar.
20	Der Motor versucht zu zünden, aber der Drehzahlanzeiger zuckt nicht. Im Motorraum tut sich auch nichts.
21	Jetzt ist es nur mittlerweile so, dass die Heizung nicht mehr funktioniert.
22	Der Motor zieht beim Beschleunigen nicht sauber durch.
23	Das Pedal der Kupplung ist schlapp. Ich kann es treten bis zum Boden.
24	Ich wollte mal wissen, wieviel der A2 im Stand so verbraucht pro Stunde.
25	Mit zunehmender Kälte hat mein Auto manchmal Probleme beim Kaltstart. Es braucht länger als gewöhnlich beim Anspringen.
26	Bei kalter Witterung springt der Motor nicht sofort an.
27	Nach Erlöschen der Heizwendel drehe ich den Schlüssel weiter auf Starten.
28	Es dauert ungefähr 20 Sekunden, bis der Anlasser den Motor startet.
29	Wenn ich nicht länger als eine halbe Stunde warte, dann startet er ganz normal.
30	Nach dem Einlegen des 1ten Gangs kann ich die Fenster nicht mehr öffnen.
31	Bei meinem A2 tritt ein rhythmisches Schlagen im Motorraum auf.
32	Wolf wurde mit seinem Auto geblitzt.

Table A.3.: All test sentences and sentence pairs for case study 1

A.3. Case Study 2 Commands

ID	Command	Pattern ID
0	Add Enrique to Development.	10
1	Add Enrique to SEDSNL.	10
2	Make Enrique supervisor for Coding.	20
3	Make Wolf responsible for ITManagement.	20
4	Make call to Wolf.	80
5	Remove Enrique from Development.	30
6	Remove Enrique from SEDSNL.	30
7	Remove task testtask from SEDSNL.	31
8	Take note for SEDSNL3.	50
9	Create project testproject1.	52
10	Create task testtask in SEDSNL.	50
11	Create entry in wiki of project SEDSNL.	51
12	Send mail to Wolf.	80
13	Send mail to all team leaders of Development.	80
14	Send mail to all team members of Development.	80
15	Send mail to all employees.	80
16	Send mail to Enrique.	80
17	Give me the chief of ITManagement.	100
18	Give me the team leader of Development.	100
19	Give me all team leaders in SEDSNL.	100
20	Give me the progress of SEDSNL.	130
21	Give me the progress from Enrique in SEDSNL.	130
22	Give me all tasks from SEDSNL.	140
23	Give me the budget of SEDSNL.	150
24	Give me the number of Enrique.	160
25	Give me all emails from Enrique.	170
26	Give me the last email from Enrique.	170
27	Give me all calls from Enrique.	170
28	Give me all mails about SEDSNL.	190
29	Give me all messages from members of development.	170
30	Give me all emails from Enrique.	170
31	Give me the last email from Enrique.	170
32	Give me the time of my last call to Enrique.	170
33	Show me my mails.	170
34	Change name of project testproject1 to oldproject1.	90
35	Change budget of SEDSNL to 100.000.	91

Table A.4.: Commands for a fictitious project management software; The row 'Pattern ID' defines the pattern which should identify this command

Acronyms

AI Artificial Intelligence

CxG Construction Grammar

DL Description Logic

DSL Domain Specific Language

FCG Fluid Construction Grammar

FOL First-Order Logic

FST Finite-State Transducer

GMF Graphical Modeling Framework

IDE Integrated Developer Environment

IE Information Extraction

IR Information Retrieval

LCA Lowest Common Ancestor

LMF Lexical Markup Framework

LTAG Lexicalized Tree Adjoining Grammar

MDD Model-driven development

N3 Notation 3

NER Named Entity Recognition

NLP Natural Language Processing

OBIE Ontology-based Information Extraction

OBIR Ontology-based Information Retrieval

OBJ Object

OBJ_2 Secondary Object

OBL Oblique Argument

OWL Web Ontology Language

OWL2 OWL 2 Web Ontology Language

PCFG Probabilistic context-free grammar

POS Part-Of-Speech

RDF Resource Description Framework

RDFS RDF Schema

SE-DSNL Semantically Enhanced Domain Specific Natural Language

SUBJ Subject

TAG Tree Adjoining Grammar

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

WSD Word Sense Disambiguation

Bibliography

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[2] Pascal Hitzler, Sebastian Rudolph, and Markus Krötzsch. *Foundations of semantic web technologies*. Chapman & Hall/CRC, 2009.

[3] Guy Deutscher. *The Unfolding of Language : An Evolutionary Tour of Mankind's Greatest Invention*. Metropolitan Books, 2005.

[4] W3C. The World Wide Web Consortium (W3C), 2006.

[5] W3C. Semantic Web: Linked Data on the Web, 2008.

[6] Daniel Hansch. Practical applications of Semantic Wikis in commercial environments. In *ISWC 2011*, 2011.

[7] Jörg Schönfisch, Florian Lautenbacher, Julian Lambertz, and Willy Chen. Softplant Living Semantic Platform. In *ISWC 2011*, 2011.

[8] Wen Zhu and Sumeet Vij. Alion Semantic Mediation Bus: An Ontology-based Runtime Infrastructure for Service Interoperability. In *ISWC 2011*, 2011.

[9] Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The semantic web revisited. *Intelligent Systems, IEEE*, 21(3):96–101, 2006.

[10] John Breslin, Alexandre Passant, and Denny Vrandec. *Handbook of Semantic Web Technologies*, volume 2. Springer Berlin Heidelberg, 2011.

[11] Pascal Hitzler, Bijan Parsia, Peter Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. *Director*, (October):1–123, 2009.

[12] Philipp Cimiano, Paul Buitelaar, John McCrae, and Michael Sintek. LexInfo: A declarative model for the lexicon-ontology interface. *Web Semantics Science Services and Agents on the World Wide Web*, 9(1):29–51, 2011.

[13] George Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[14] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*, volume 71 of *Language, Speech, and Communication*. MIT Press, 1998.

[15] Wolf Fischer and Bauer Bernhard. Cognitive-Linguistics-based Request Answer System. In *Adaptive Multimedia Retrieval. Understanding Media and Adapting to the User*, pages 135–146, Madrid, 2009. Springer.

[16] Wolf Fischer and Bernhard Bauer. Domain Dependent Semantic Requirement Engineering. *Domain Engineering DE @ CAiSE'2010*, page 6, 2010.

[17] Wolf Fischer and Bernhard Bauer. Combining Ontologies And Natural Language. *Proceedings of the Sixth Australasian Ontology Workshop*, 2010.

[18] Wolf Fischer and Bernhard Bauer. Ontology based Spreading Activation for NLP related Scenarios. In *SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing*, pages 56–61, 2011.

[19] Adrian Akmajian, Richard A Demers, Ann K Farmer, and Robert M Harnish. *Linguistics: An Introduction to Language and Communication*. The MIT Press, 5th editio edition, 2001.

[20] Dan Jurafsky and James Martin. *Speech and Language Processing*. Prentice Hall International; Auflage: 2nd edition., 2009.

[21] Ruslan Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford Handbooks in Linguistics. Oxford University Press, 2003.

[22] Halvor Eifring and Rolf Theil. Linguistics for Students of Asian and African Languages, 2005.

[23] Noam Chomsky. *Syntactic Structures*, volume 7. Mouton, 2004.

[24] Benjamin W Fortson. *Indo-European Language and Culture: An Introduction (Blackwell Textbooks in Linguistics)*. Wiley-Blackwell, 2004.

[25] Mark Aronoff, Janie Rees-miller, Lyle Campbell, Bob Carpenter, and Abigail Cohn. The Handbook of Linguistics. In Mark Aronoff and Janie Rees-Miller, editors, *The Handbook of Linguistics*, volume 43, chapter 12, page 824. Blackwell, 2002.

[26] Elmar Kremer. The Stanford Encyclopedia of Philosophy: Antoine Arnauld, 2012.

[27] Antoine Arnauld and Claude Lancelot. *A general and rational grammar*. Scolar Press, 1968.

[28] Paul R Kroeger. *Analyzing grammar: An introduction*. Cambridge University Press, 2005.

[29] Jan Rijkhoff. When can a language have nouns and verbs? *Acta Linguistica Hafnien-*

sia, 35(1):7–38, 2003.

[30] Mark C Baker. *Lexical categories: Verbs, nouns and adjectives*, volume 102. Cambridge University Press, 2003.

[31] Charles J Fillmore. FRAME SEMANTICS AND THE NATURE OF LANGUAGE. *Annals of the New York Academy of Sciences*, 280(1):20–32, 1976.

[32] Christopher D Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[33] The Cambridge Grammar and English Language. *A student's introduction to English grammar*, volume 84. Cambridge University Press, 2008.

[34] L T F Gamut. *Logic, Language and Meaning, volume 1: Introduction to Logic*, volume 1 of *Lecture Notes in Computer Science*. University of Chicago Press, 1991.

[35] Donald Davidson. Truth and meaning. *Synthese*, 17(1):304–323, 1967.

[36] G Frege. Ausführungen über Sinn und Bedeutung. *Nachgelassene Schriften*, pages 128–135, 1892.

[37] Ludwig Wittgenstein. Philosophical Investigations. 2009.

[38] Adam Sennet. Ambiguity. In Edward N Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 201 edition, 2011.

[39] George Dunbar. Towards a cognitive analysis of polysemy, ambiguity, and vagueness. *Cognitive Linguistics*, 12(1):1–14, 2001.

[40] Gregory Ward. Equatives and Deferred Reference. *Language*, 80(2):262–289, 2004.

[41] G Nunberg. Transfers of meaning. *Journal of semantics*, 12(2):109–132, 1995.

[42] William Croft and D Alan Cruse. *Cognitive Linguistics*, volume 3 of *Cambridge Textbooks in Linguistics*. Cambridge University Press, 2004.

[43] Peter Robinson and Nick C Ellis. *Handbook of cognitive linguistics and second language acquisition*. Routledge, 2008.

[44] Theo Janssen and Gisela Redeker. *Cognitive Linguistics, Foundations, Scope, and Methodology*. Mouton de Gruyter, 1999.

[45] Luc Steels and Joachim De Beule. A (very) brief introduction to fluid construction grammar. In *Proceedings of the Third Workshop on Scalable Natural Language Understanding*, pages 73–80. Association for Computational Linguistics, 2006.

[46] Luc Steels and Joachim De Beule. Unify and merge in fluid construction grammar. *Lecture Notes in Computer Science*, 4211:197, 2006.

[47] Luc Steels. Fluid Construction Grammars. A Brief Tutorial. 2004.

[48] Ronald W Langacker. *Cognitive Grammar*, volume 1 of *Current Issues in Linguistic Theories*. Oxford University Press, 2002.

[49] Benjamin K Bergen and Nancy Chang. Embodied construction grammar in simulation-based language understanding. *Construction grammars: Cognitive grounding and theoretical extensions*, pages 147–190, 2005.

[50] William Croft. *Radical construction grammar: Syntactic theory in typological perspective*. Oxford University Press, USA, 2001.

[51] Ronald W Langacker. An introduction to cognitive grammar. *Cognitive Science: A Multidisciplinary Journal*, 10(1):1–40, 1986.

[52] Martin Kay. Functional unification grammar: a formalism for machine translation. In *Proc of COLING 84*, pages 75–78. North-Holland, 1984.

[53] Claude E Shannon and Warren Weaver. *Mathematical Theory of Communication*, volume 27. University of Illinois Press, 1949.

[54] Robert Dale, Hermann Moisl, and Harold Somers. *Handbook of Natural Language Processing*. CRC Press, 2000.

[55] Aravind K Joshi. An introduction to tree adjoining grammars. *Mathematics of language*, 1:87–115, 1987.

[56] Carl Pollard and Ivan A Sag. *Head-Driven Phrase Structure Grammar*, volume 72 of *Stauffenburg-Einführungen*. University of Chicago Press, 1994.

[57] Raúl Rojas. *Neural networks: a systematic introduction*. Springer, 1996.

[58] Simon Haykin. *Neural Networks: A Comprehensive Foundation*, volume 13. Prentice Hall, 1999.

[59] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[60] L R Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[61] Eric Brill. A simple rule-based part of speech tagger. *Proceedings of the third conference on Applied natural language processing*, 28(4):152–155, 1992.

[62] Roger Garside. The CLAWS word-tagging system. *The Computational Analysis of English: A Corpus-based Approach. London: Longman*, pages 30–41, 1987.

[63] Geoffrey Leech, Roger Garside, and Michael Bryant. CLAWS4: the tagging of the British National Corpus. In *Proceedings of the 15th conference on Computational linguistics-Volume 1*, pages 622–628. Association for Computational Linguistics, 1994.

[64] Kristina Toutanova and Christopher D Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics*, 13(121):63–70, 2000.

[65] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology NAACL 03*, 1(June):173–180, 2003.

[66] Libin Shen, Giorgio Satta, and Aravind K Joshi. Guided Learning for Bidirectional Sequence Classification. In *Computational Linguistics*, volume 45, pages 760–767. Association for Computational Linguistics, 2007.

[67] Drahomíra Johanka Spoustová, Jan Hajič, Jan Raab, and Miroslav Spousta. Semi-supervised training for the averaged perceptron POS tagger. *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics on EACL 09*, 12(April):763–771, 2009.

[68] Anders Sogaard. Simple semi-supervised training of part-of-speech taggers. In *Proceedings of the ACL 2010 Conference Short Papers*, number July, pages 205–208. Association for Computational Linguistics, 2010.

[69] Eugenie Giesbrecht and Stefan Evert. Is Part-of-Speech Tagging a Solved Task? An Evaluation of POS Taggers for the German Web as Corpus. *Web as Corpus Workshop WAC5*, page 27, 2009.

[70] Christopher D Manning. Part-of-speech tagging from 97% to 100%: is it time for some linguistics? *Computational Linguistics and Intelligent Text Processing*, pages 171–189, 2011.

[71] Hermann Ney. Dynamic programming parsing for context-free grammars in continuous speech recognition, 1991.

[72] Ann Taylor, Mitchell P Marcus, and Beatrice Santorini. Chapter 1 THE PENN TREEBANK : AN OVERVIEW. In Anne Abeille, editor, *Treebanks Building and Using Parsed Corpora*, chapter 1, pages 5–22. Kluwer Academic Publishers, 2003.

[73] J K Baker. Trainable Grammars for Speech Recognition. *Proceedings of the Spring*

Conference of the Acoustical Society of America, 65(S1):547–550, 1979.

[74] Elisabet Comelles, Gran Via, Corts Catalanes, Victoria Arranz, Brillat Savarin, and Irene Castellón. Constituency and Dependency Parsers Evaluation. pages 59–66, 2010.

[75] Daniel Cer, Marie-Catherine De Marneffe, Daniel Jurafsky, and Christopher D Manning. Parsing to Stanford Dependencies: Trade-offs between speed and accuracy. In *Proceedings of LREC*, volume 0, pages 1628–1632, 2010.

[76] Yusuke Miyao, Rune Sæ tre, Kenji Sagae, and Takuya Matsuzaki. Task-oriented Evaluation of Syntactic Parsers and Their Representations. *Computational Linguistics*, (June):46–54, 2008.

[77] Jerrold J Katz and Jerry A Fodor. The structure of a semantic theory. *Language*, 39(2):170–210, 1963.

[78] Yorick A Wilks. *Grammar, meaning and the machine analysis of language*. Routledge & Kegan Paul London, 1972.

[79] Graeme Hirst. *Semantic interpretation and the resolution of ambiguity*. Cambridge University Press, 1992.

[80] Nancy M Ide and Jean Veronis. Mapping dictionaries: A spreading activation approach. In *6th Annual Conference of the Centre for the New Oxford English Dictionary*, pages 52–64, 1990.

[81] Peter F Brown, Stephen A Della Pietra, Vincent J Della Pietra, and Robert L Mercer. Word-sense disambiguation using statistical methods. In *Proceedings of the 29th annual meeting on Association for Computational Linguistics*, pages 264–270. Association for Computational Linguistics, 1991.

[82] Rodney Huddleston. Introduction to the Grammar of English. *Language*, 63(3):635, 1984.

[83] Jerry R Hobbs. Resolving pronoun references. *Lingua*, 44(4):311–338, 1978.

[84] Barbara J Grosz, Scott Weinstein, and Aravind K Joshi. Centering : A Framework for Modeling the Local Coherence of Discourse. *Computational Linguistics*, 21(2):203–225, 1995.

[85] Bonnie L Webber. A formal approach to discourse anaphora. Technical report, DTIC Document, 1978.

[86] Thomas Hofweber. Logic and Ontology. In Edward N Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 201 edition, 2012.

[87] Tom Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928, 1995.

[88] Heiner Stuckenschmidt. *Ontologien: Konzepte, Technologien und Anwendungen*. Informatik im Fokus. Springer, 2009.

[89] Frank Manola and Eric Miller. RDF Primer, 2004.

[90] Tim Berners-Lee. Notation3 (N3) A readable RDF syntax, 1998.

[91] Dan Brickley and R V Guha. RDF Vocabulary Description Language 1.0: RDF Schema, 2004.

[92] Sean Bechhofer, Frank Van Harmelen, Jim Hendler, Ian Horrocks, Deborah L McGuinness, Peter F Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference, 2004.

[93] Deborah L McGuinness and Frank Van Harmelen. OWL Web Ontology Language Overview. *W3C recommendation*, 10(February):1–22, 2004.

[94] OWL 2 Web Ontology Language Document Overview. *October*, 2(October):1–12, 2009.

[95] Alistair Miles and Sean Bechhofer. SKOS Simple Knowledge Organization System Reference, 2008.

[96] Alistair Miles, Brian Matthews, Dave Beckett, Dan Brickley, Michael Wilson, and Nikki Rogers. SKOS: A language to describe simple knowledge structures for the web. In *XTech 2005 Conference Proceedings*, 2005.

[97] Gil Francopoulo, Monte George, Nicoletta Calzolari, Monica Monachini, Nuria Bel, Mandy Pet, and Claudia Soria. Lexical markup framework (LMF). In *International Conference on Language Resources and Evaluation-LREC 2006*, 2006.

[98] Gil Francopoulo, Nuria Bel, Monte George, Nicoletta Calzolari, Monica Monachini, Mandy Pet, and Claudia Soria. Lexical markup framework: ISO standard for semantic information in NLP lexicons. In *Proceedings of the Workshop of the GLDV Working Group on Lexicography at the Biennial Spring Conference of the GLDV*, 2007.

[99] L M F Working Group. Language resource management-Lexical markup framework (LMF). Technical report, Technical Report ISO/TC 37/SC 4, 2008.

[100] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*, volume 1. Cambridge University Press, 2008.

[101] Gerald Kowalski. Information Retrieval Architecture and Algorithms. *Information Retrieval*, pages 253–281, 2011.

[102] Jim Cowie and Yorick Wilks. Information extraction. *Handbook of Natural Language Processing*, pages 241–260, 2000.

[103] *MUC4 '92: Proceedings of the 4th conference on Message understanding*, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.

[104] Daya C Wimalasuriya and Dejing Dou. Ontology-based information extraction: An introduction and a survey of current approaches. *Journal of Information Science*, 36(3):306–323, 2010.

[105] David W Embley. Toward semantic understanding: an approach based on information extraction ontologies. *Science*, 27(Winslett):3–12, 2004.

[106] Alexander Maedche and Steffen Staab. Bootstrapping an Ontology-based Information Extraction System. *Machine Learning*, 111:345–360, 2002.

[107] Burcu Yildiz and Silvia Miksch. ontoX - A Method for Ontology-Driven Information Extraction. *Main*, 4707, Part:1–14, 2007.

[108] Paul Buitelaar, Philipp Cimiano, Peter Haase, and Michael Sintek. Towards linguistically grounded ontologies. In *Proceedings of the 6th European Semantic Web Conference (ESWC09)*, pages 111–125. Springer, 2009.

[109] John Ridley Stroop. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 18(6):643–662, 1935.

[110] John A Bargh, Mark Chen, and Lara Burrows. Automaticity of social behavior: Direct effects of trait construct and stereotype activation on action. *Journal of personality and social psychology*, 71:230–244, 1996.

[111] Natalya F Noy and Deborah L Mcguinness. Ontology Development 101 : A Guide to Creating Your First Ontology. *Development*, 32(1):1–25, 2000.

[112] Protégé. Protege Ontology Library.

[113] Christian Saad and Bernhard Bauer. Data-flow Based Model Analysis. *Second NASA Formal Methods Symposium accepted*, pages 227–231, 2010.

[114] Brooke Cowan and Michael Collins. Morphology and reranking for the statistical parsing of Spanish. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing HLT 05*, number October, pages 795–802. Association for Computational Linguistics, 2005.

[115] Reut Tsarfaty and Khalil Sima'an. Relational-realizational parsing. In *Computational Linguistics*, volume 1, pages 889–896. Association for Computational Linguistics, 2008.

[116] Michael Collins, Jan Hajič, Eric Brill, Lance Ramshaw, and Christoph Tillmann. A Statistical Parser of Czech. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics ACL 99*, pages 505–512. Association for Computational Linguistics, 1999.

[117] John A Haywood and H M Nahmad. *A new Arabic grammar of the written language*, volume 48. Harvard University Press, 1962.

[118] Sandi Pohorec, Ines Ceh, Milan Zorman, Marjan Mernik, and Peter Kokol. Natural language processing resources: Using semantic web technologies.

[119] Elena Montiel-ponsoda, Guadalupe Aguado De Cea, and Asunción Gómez-pérez. Modelling multilinguality in ontologies. *Coling 2008 Companion volume Posters*, (August):67–70, 2008.

[120] W Peters, Elena Montiel-Ponsoda, G Aguado de Cea, and A Gómez-Pérez. Localizing Ontologies in OWL, November 2007.

[121] John Bateman, Bernardo Magnini, and Giovanni Fabris. The generalized upper model knowledge base: Organization and use. *Towards very large knowledge bases*, pages 60–72, 1995.

[122] John Bateman, Renate Henschel, and Fabio Rinaldi. The generalized upper model 2.0. In *Proceedings of the ECAI94 Workshop: Comparison of Implemented Ontologies*, 1995.

[123] J A Bateman, R T Kasper, J D Moore, and R A Whitney. A general organization of knowledge for natural language processing: the penman upper model. Technical report, Technical report, USC/Information Sciences Institute, Marina del Rey, CA, 1990.

[124] Renate Henschel and John Bateman. The merged upper model: a linguistic ontology for German and English. In *Proceedings of the 15th conference on Computational linguistics-Volume 2*, pages 803–809. Association for Computational Linguistics, 1994.

[125] P Cimiano, P Haase, M Herold, M Mantel, and P Buitelaar. LexOnto: A Model for Ontology Lexicons for Ontology-based NLP. In *Proc. OntoLex07 Workshop*. Citeseer, 2007.

[126] Paul Buitelaar, Thierry Declerck, Anette Frank, Stefania Racioppa, Malte Kiesel, Michael Sintek, Ralf Engel, Massimo Romanelli, Daniel Sonntag, Berenike Loos, Vanessa Micelli, Robert Porzel, and Philipp Cimiano. Linginfo: Design and Applications of a Model for the Integration of Linguistic Information in Ontologies. In *Proceedings of OntoLex 2006*. Citeseer, 2006.

[127] John McCrae, Dennis Spohr, and Philipp Cimiano. Linking lexical resources and ontologies on the semantic web with lemon. *Lecture Notes in Computer Science including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*, 6643 LNCS(PART 1):245–259, 2011.

[128] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*, volume 39. Addison-Wesley, 2006.

[129] Vitaly I Voloshin. *Introduction to Graph and Hypergraph Theory*. Nova Kroshka Books, 2009.

[130] Eric Bengtson and Dan Roth. Understanding the value of features for coreference resolution. *Proceedings of the Conference on Empirical Methods in Natural Language Processing EMNLP 08*, 51(October):294–303, 2008.

[131] Christina Unger and Philipp Cimiano. Representing and resolving ambiguities in ontology-based question answering. In *Proceedings of the TextInfer 2011 Workshop on Textual Entailment*, pages 40–49. Association for Computational Linguistics, 2011.

[132] Philipp. Cimiano and Uwe Reyle. Ontology-based semantic construction, underspecification and disambiguation. In *Proceedings of the Prospects and Advances in the Syntax-Semantic Interface Workshop*, pages 33–38, 2003.

[133] Jon Curtis, John Cabral, and David Baxter. On the application of the cyc ontology to word sense disambiguation. In *Proceedings of the Nineteenth International FLAIRS Conference. Melbourne Beach, FL, USA: AAAI Press*, pages 652–657, 2006.

[134] John A Bateman. The Theoretical Status of Ontologies in Natural Language Processing. *Arxiv preprint cmplg9704010*, pages 1–43, 1997.

[135] David Vallet, Miriam Fernandez, and Pablo Castells. An Ontology-Based Information Retrieval Model. *The Semantic Web Research and Applications*, 3532(5):455–470, 2005.

[136] Atanas Kiryakov, Borislav Popov, Ivan Terziev, Dimitar Manov, and Damyan Ognyanoff. Semantic annotation, indexing, and retrieval. *Web Semantics Science Services and Agents on the World Wide Web*, 2(1):49–79, 2004.

[137] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. *Text Processing with GATE (Version 6)*. 2011.

[138] Wei Zhou, Clement Yu, Neil Smalheiser, Vetle Torvik, and Jie Hong. Knowledge-intensive conceptual retrieval and passage extraction of biomedical literature. *Pro-*

ceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval SIGIR 07, page 655, 2007.

[139] Zhenyu Liu and Wesley W Chu. Knowledge-based query expansion to support scenario-specific retrieval of medical free text. *Information Retrieval*, 10(2):173–202, 2007.

[140] Bevan Koopman, Peter D Bruza, Laurianne Sitbon, and Michael Lawley. Towards Semantic Search and Inference in Electronic Medical Records: an approach using Concept-based Information Retrieval. *Science And Technology*, In Press:1–11, 2011.

[141] Kent A Spackman and Keith E Campbell. Compositional concept representation using SNOMED: towards further convergence of clinical terminologies. *Proceedings of the AMIA Symposium*, pages 740–744, 1998.

[142] Alan R Aronson and François-Michel Lang. An overview of MetaMap: historical perspective and recent advances. *Journal of the American Medical Informatics Association : JAMIA*, 17(3):229–236, 2010.

[143] Rose Dieng-Kuntz, Inria Sophia Antipolis, Pascal Barbry, and Sophia Antipolis. An ontology-based approach to support text mining and information retrieval in the biological domain. *Computer*, 13(12):1881–1907, 2007.

[144] Helmut Schmid. Probabilistic Part-of-Speech Tagging Using Decision Trees. In Daniel Jones, editor, *Proceedings of International Conference on New Methods in Language Processing*, volume 12 of *Studies in Computational Linguistics*, pages 44–49. University of Stuttgart, Citeseer, 1994.

[145] Ted Briscoe and John Carroll. Robust accurate statistical annotation of general text. *SciencesNew York*, pages 1499–1504, 2002.

[146] Hans-Michael Müller, Eimear E Kenny, and Paul W Sternberg. Textpresso: an ontology-based information retrieval and extraction system for biological literature. *PLoS Biology*, 2(11):e309, 2004.

[147] Hans-Michael Müller, Arun Rangarajan, Tracy K Teal, and Paul W Sternberg. Textpresso for neuroscience: searching the full text of thousands of neuroscience research papers. *Neuroinformatics*, 6(3):195–204, 2008.

[148] Iulian-Florin Toma. Using ontologies as queries in information retrieval, 2011.

[149] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[150] Jacob Köhler, Stephan Philippi, Michael Specht, and Alexander Rüegg. Ontology based text indexing and querying for the semantic web. *Knowledge-Based Systems*,

19(8):744–754, 2006.

[151] Paul Kogut and William Holmes. AeroDAML: Applying Information Extraction to Generate DAML Annotations from Web Pages. In Siegfried Handschuh, R Dieng, and Steffen Staab, editors, *Database*, page 3. CEUR Workshop Proceedings, 2001.

[152] Paul Kogut. AeroDAML Applying Information Extraction to Generate DAML Annotations, 2001.

[153] Alexiei Dingli, Fabio Ciravegna, and Yorick Wilks. Automatic semantic annotation using unsupervised information extraction and integration. In *Proceedings of SemAnnot 2003 Workshop*, 2003.

[154] Fabio Ciravegna, Sam Chapman, Alexiei Dingli, and Yorick Wilks. Learning to Harvest Information for the Semantic Web. *Learning*, 3053(1):312–326, 2004.

[155] Sam Chapman, Barry Norton, and Fabio Ciravegna. Armadillo: Integrating knowledge for the semantic web. In *Proceedings of the Dagstuhl Seminar in Machine Learning for the Semantic Web*, 2005.

[156] Borislav Popov, Atanas Kiryakov, Angel Kirilov, and Dimitar Manov. KIM âĂŞ Semantic Annotation Platform. *Engineering*, 2870(3-4):834–849, 2003.

[157] Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A Tomlin, Jason Y Zien, Harry Road, and San Jose. SemTag and Seeker: Bootstrapping the semantic web via automated semantic annotation. *Proceedings of the 12th international conference on World Wide Web*, 1(Section 2):178–186, 2003.

[158] Siegfried Handschuh, Steffen Staab, and Fabio Ciravegna. S-CREAM - Semi-automatic CREAtion of Metadata. *Proceedings of EKAW 2002*, LNAI 2473:358–372, 2002.

[159] Daya C Wimalasuriya. Use of ontologies in information extraction. 2011.

[160] David Ferrucci, Eric Brown, Jennifer Chu-carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building Watson : An Overview of the DeepQA Project. *AI Magazine*, 31(3):59–79, 2010.

[161] Philipp Cimiano and Johanna Völker. Text2Onto - A Framework for Ontology Learning and Data-driven Change Discovery. In Andres Montoyo, Rafael Munoz, and Elisabeth Metais, editors, *Proceedings of the 10th International Conference on Applications of Natural Language to Information SystemsNLDB*, volume 3513 of *LNCS*, pages 227–238. Springer, 2005.

[162] Hamish Cunningham, Diana Maynard, and Valentin Tablan. JAPE: a Java Annotation Patterns Engine (Second Edition). (Technical Report CS–00–10), 2000.

[163] Amalia Todirascu, Laurent Romary, and Dalila Bekhouche. Vulcain - An Ontology-Based Information Extraction System. *Natural Language Processing and Information Systems*, pages 64–75, 2002.

[164] Paul Buitelaar, Philipp Cimiano, Anette Frank, and Stefania Racioppa. SOBA : SmartWeb Ontology-based Annotation. *Proceedings of the Demo Session at the International Semantic Web Conference*, 2006.

[165] Witold Drozdzynski, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. Shallow Processing with Unification and Typed Feature Structures — Foundations and Applications. *Künstliche Intelligenz*, 1(1):17–23, 2004.

[166] Benjamin Adrian, Jörn Hees, Ludger Van Elst, and Andreas Dengel. iDocument: Using Ontologies for Extracting and Annotating Information from Unstructured Text. *KI 2009 Advances in Artificial Intelligence*, 5803(2002):249–256, 2009.

[167] Benjamin Adrian, Heiko Maus, Malte Kiesel, and Andreas Dengel. Towards ontology-based information extraction and annotation of paper documents for personalized knowledge acquisition. *Proceedings of the First International Workshop on Personal Knowledge Management*, 2009.

[168] Christina Unger and Philipp Cimiano. Pythia: Compositional meaning construction for ontology-based question answering on the Semantic Web. *Natural Language Processing and Information Systems*, pages 153–160, 2011.

[169] Philipp Cimiano. Flexible semantic composition with DUDES. In *Proceedings of the Eighth International Conference on Computational Semantics*, pages 272–276. Association for Computational Linguistics, 2009.

[170] Yves Schabes and Aravind K Joshi. An Earley-type parsing algorithm for tree adjoining grammars. In *Proceedings of the 31th Annual Meeting of the Association for Computational Linguistics*, pages 258–269. Association for Computational Linguistics, 1988.

[171] Yaoyong Li and Kalina Bontcheva. Hierarchical , Perceptron-like Learning for Ontology-Based Information Extraction. *Learning*, 23(1):777–786, 2007.

[172] Ofer Dekel, Joseph Keshet, and Yoram Singer. Large Margin Hierarchical Classification. *Journal of the American Statistical Association*, 104(487):906–909, 2004.

[173] Fei Wu, Raphael Hoffmann, and Daniel S Weld. Information extraction from Wikipedia: Moving down the long tail. In *Proceeding of the 14th ACM SIGKDD in-*

ternational conference on Knowledge discovery and data mining, pages 731–739. ACM, 2008.

[174] Fei Wu and Daniel S Weld. Open information extraction using Wikipedia. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 118–127. Association for Computational Linguistics, 2010.

[175] Luke K Mcdowell and Michael Cafarella. Ontology-Driven Information Extraction with OntoSyphon. *International Semantic Web Conference*, 4273(2):428–444, 2006.

[176] Philipp Cimiano, Siegfried Handschuh, and Steffen Staab. Towards the self-annotating web. *Proceedings of the 13th conference on World Wide Web WWW 04*, 462-471:462, 2004.

[177] Horacio Saggion, Adam Funk, Diana Maynard, and Kalina Bontcheva. Ontology-based Information Extraction for Business Intelligence. *Intelligence*, 4825:843–856, 2007.

[178] Michal Laclavik, Martin Seleng, Marek Ciglan, and Ladislav Hluchy. ONTEA: PLATFORM FOR PATTERN BASED AUTOMATED SEMANTIC ANNOTATION. *Computing and Informatics*, 28(May):555–579, 2009.

[179] Chung Hee Hwang. Incompletely and Imprecisely Speaking : Using Dynamic Ontologies for Representing and Retrieving Information. *Trains*, (Mcc):14–20, 1999.

[180] Maria Vargas-Vera, Enrico Motta, John Domingue, Simon Buckingham Shum, and Mattia Lanzoni. Knowledge extraction by using an ontology-based annotation tool. *Knowledge Creation Diffusion Utilization*, pages 1–8, 2001.

[181] Sven J Körner and Mathias Landhäuß er. Semantic Enriching of Natural Language Texts with Automatic Thematic Role Annotation. In *Proceedings of the Natural language processing and information systems, and 15th international conference on Applications of natural language to information systems*, Heidelberg, 2010. Springer Berlin / Heidelberg.

[182] Tom Gelhausen and Walter F Tichy. Thematic Role Based Generation of UML Models from Real World Requirements, 2007.

[183] The Stanford NLP (Natural Language Processing) Group. http://nlp.stanford.edu/.

[184] ResearchCyc. http://research.cyc.com/.

[185] Martin Dzbor, John Domingue, and Enrico Motta. Magpie - towards a semantic web browser. *Scenario*, 2870:690–705, 2003.

[186] John Domingue, Martin Dzbor, and Enrico Motta. Magpie: supporting browsing and navigating on the semantic web. *Human Factors*, pages 191–197, 2004.

[187] Laurian Gridinoc, Marta Sabou, Mathieu D'Aquin, Martin Dzbor, and Enrico Motta. Semantic browsing with PowerMagpie. *The Semantic Web: Research and Applications*, pages 802–806, 2008.

[188] John McCrae, Elena Montiel-Ponsoda, and Philipp Cimiano. Integrating WordNet and Wiktionary with lemon. *Linked Data in Linguistics*, pages 25–34, 2012.

[189] Brian Davis, Fadi Badra, Paul Buitelaar, Tobias Wunner, and Siegfried Handschuh. Squeezing lemon with GATE. *MSW 2011*, page 74, 2011.

[190] M Ross Quillian. A revised design for an understanding machine. *Mechanical translation*, 7(1):17–29, 1962.

[191] M Ross Quillian. Semantic memory, 1968.

[192] Allan M Collins and Elizabeth F Loftus. A spreading-activation theory of semantic processing. *Psychological Review*, 82(6):407–428, 1975.

[193] Ulrich Schiel. Abstractions in semantic networks: axiom schemata for generalization, aggregation and grouping. *ACM SIGART Bulletin*, (107):25–26, 1989.

[194] Rada Mihalcea and Dragomir Radev. *Graph-based natural language processing and information retrieval*. Cambridge University Press, 2011.

[195] Joachim Kleb and Andreas Abecker. Entity Reference Resolution via Spreading Activation on RDF-Graphs. *The Semantic Web Research and Applications*, 6088:152–166, 2010.

[196] George Tsatsaronis, Michalis Vazirgiannis, and Ion Androutsopoulos. Word Sense Disambiguation with Spreading Activation Networks Generated from Thesauri. In Manuela M Veloso, editor, *IJCAI 2007*, pages 1725–1730, 2007.

[197] George Tsatsaronis, Iraklis Varlamis, and Michalis Vazirgiannis. Word Sense Disambiguation with Semantic Networks. *Work*, 5246:219–226, 2008.

[198] George Tsatsaronis, Iraklis Varlamis, and Kjetil Nø rvåg. An experimental study on unsupervised graph-based word sense disambiguation. *Computational Linguistics and Intelligent Text Processing*, pages 184–198, 2010.

[199] Eneko Agirre, Aitor Soroa, and Mark Stevenson. Graph-based word sense disambiguation of biomedical documents. *Bioinformatics*, 26(22):2889–2896, 2010.

[200] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *World Wide Web Internet And Web*

Information Systems, 54(2):1–17, 1998.

[201] Sin-Jae Kang and Jong-Hyeok Lee. Ontology-based word sense disambiguation using semi-automatically constructed ontology. *MT Summit VIII Machine Translation in the Information Age Proceedings Santiago de Compostela Spain 1822 September 2001 pp181186 PDF 287KB*, 2001.

[202] Tim Hussein, Daniel Westheide, and Jürgen Ziegler. Context-adaptation based on Ontologies and Spreading Activation. *Citeseer*, 2005.

[203] Akrivi Katifori, Costas Vassilakis, and Alan Dix. Ontologies and the brain: Using spreading activation through ontologies to support personal interaction. *Cognitive Systems Research*, 11(1):25–41, 2010.

[204] Alan Dix. The brain and the web-intelligent interactions from the desktop to the world. *Simpósio de Fatores Humanos em Sistemas Computacionais (IHC 2006)*, 2006.

[205] Andy Seaborne. RDQL - A Query Language for RDF (Member Submission), 2004.

[206] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF, 2008.

[207] Apache Jena Property Page. http://jena.sourceforge.net/ARQ/property_paths.html, 2012.

[208] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. *Proceedings of the 1st Conference on Social Semantic Web CSSW*, 221(ISBN 978-3-88579-207-9):59–68, 2007.

[209] Apache Jena ARQ Counting. http://jena.apache.org/documentation/query/group-by.html, 2012.

[210] Suphakit Niwattanakul, Philippe Martin, Michel Eboueya, and Kanit Khaimook. Ontology Mapping based on Similarity Measure and Fuzzy Logic. In *World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education*, volume 2007, pages 6383–6387, 2007.

[211] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Introduction to Data Mining. *Journal of School Psychology*, 19(1):51–56, 2005.

[212] Tuukka Ruotsalo. Domain specific data retrieval on the semantic web. *The Semantic Web: Research and Applications*, pages 422–436, 2012.

[213] G Salton, A Wong, and C S Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[214] Miriam Fernández Sánchez. *Semantically enhanced Information Retrieval : an ontology-*

based approach. Dissertation, Universidad Autonoma De Madrid, 2009.

[215] Pablo Castells, Miriam Fernandez, and David Vallet. An Adaptation of the Vector-Space Model for Ontology-Based Information Retrieval, 2007.

[216] Bettina Fazzinga, Giorgio Gianforme, Georg Gottlob, and Thomas Lukasiewicz. Semantic Web Search Based on Ontological Conjunctive Queries. *Web Semantics Science Services and Agents on the World Wide Web*, 9(4):453–473, 2011.

[217] Xiaomin Ning, Hai Jin, Weijia Jia, and Pingpeng Yuan. Practical and effective IR-style keyword search over semantic web. *Information Processing & Management*, 45(2):263–271, 2009.

[218] Antonio M Rinaldi. An ontology-driven approach for semantic information retrieval on the Web. *ACM Trans Internet Technol*, 9(3):1–24, 2009.

[219] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design Science Research in Information Systems. *Information Systems Journal*, 22(1):75–105, 2004.

[220] Hong Zhu. *Software Design Methodology: From Principles to Architectural Styles*. Butterworth-Heinemann, 2005.

[221] Hoifung Poon and Lucy Vanderwende. Joint Inference for Knowledge Extraction from Biomedical Literature. *Computational Linguistics*, (June):813–821, 2010.

[222] Hoifung Poon and Pedro Domingos. Joint Inference in Information Extraction. In *Artificial Intelligence*, volume 22, pages 913–918. AAAI Press, 2007.

[223] Violaine Prince and Gerard Sabah. Coping with vague and fuzzy words: A multi-expert natural language system which overcomes ambiguities. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI-92)*. Citeseer, 1992.

[224] Jeff Howe. The Rise of Crowdsourcing. *North*, 14(14):1–5, 2006.

[225] D C Brabham. Crowdsourcing as a Model for Problem Solving: An Introduction and Cases. *Convergence: The International Journal of Research into New Media Technologies*, 14(1):75–90, 2008.

[226] Mehryar Mohri. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2):269–311, 1997.

[227] Eclipse. Graphical Modeling Project (GMP).

[228] Beatrice Santorini. Part-of-speech tagging guidelines for the Penn Treebank Project (3rd revision). 1990.

List of Figures

1.1. The Semantic Web Stack according to the W3C [5] 3
1.2. Architecture of the SE-DSNL approach 14
1.3. Outline of the thesis 18

2.1. An example of a constituent tree, which has been generated with the Stanford Parser 2.0.1 and the English PCFG caseless model 24
2.2. Some varieties of meaning [19] 27
2.3. Example of a lexical stem rule [47] 31
2.4. Example of a semantic categorization rule [47] 31
2.5. Semantic result for the sentence "Jill slides blocks to Jack" [47] 32
2.6. Construction for mapping a sentence, containing a sentence-verb-object-to-object syntactic structure, to a transfer-to-target semantic structure [47] 33
2.7. Simple english grammar [20] 35
2.8. Syntax tree of the sentence "Book the flight" [20] 35
2.9. Example of an ontology containing the basic elements 41
2.10. An RDF graph describing a person with the name Eric Miller [89] 42
2.11. Representation of the Eric Miller RDF Graph in N3 form 43
2.12. Representation of the Eric Miller RDF Graph in N3 form 44
2.13. Abstract overview of LexInfo [12] 48
2.14. Syntactic extension of Lexical Markup Framework [12] 48
2.15. Semantic extension of Lexical Markup Framework [12] 49
2.16. Associating the lemma "river" with some of its morphological variations [12] 49
2.17. Decomposition of the german word "Autobahnkreuz" into its single components and mapping to ontological elements [12] 50
2.18. Representation of the syntactic behaviour of "flow" and its mapping to the ontology [12] 51
2.19. Functional overview of an IR system [101] 52
2.20. General architecture of an OBIE system [104] 55

3.1. Overall approach for the design of the meta model 63
3.2. Overview of the meta model structure 64

3.3. Content of the default Scope of the meta model 65
3.4. Representation of the sentence "The detective assumes that the gardner killed the man" 66
3.5. Semantic Scope 66
3.6. Example of the structure for "Person lives in House" 67
3.7. Syntactic Scope 68
3.8. Construction Scope 69
3.9. Interpretation Scope 70
3.10. 'University' Construction Example 83
3.11. 'University of Augsburg' Construction Example 85
3.12. NP → VP → NP Construction Example 87
3.13. Construction leading to an infinite loop 91

4.1. Syntax trees for the two sentences "When I go of the cludge of my Polo, the engine just stops. Yet it stutters.", created with Stanford Parser V1.6.8 and English PCFG grammar 100
4.2. Syntax tree of the modified first sentence ('just' has been removed), created with Stanford Parser V1.6.8 and English factored grammar 101
4.3. Example of different possible Constructions matching a node in a syntax tree 102
4.4. Example of different possible Constructions matching a wrong syntax tree 102
4.5. Possibilities to match a two argument Construction to a syntax tree 103
4.6. Construction instances and Construction instance relation 108
4.7. Levels of different instances 113
4.8. Example of a hypergraph structure for a single sentence 114
4.9. Example of the hypergraph structure for multiple sentences 115
4.10. Syntax tree with two overlapping Constructions instances; The blue colored instance additionally references node N_2, whereas the other instance would reference N_3 127
4.11. 'InOrder' Function optimizing the *elems* list copies based on the position of the different entries 128
4.12. Pronominal Anaphor Resolution Concept Overview 130
4.13. Two Syntax Trees as an Example for Distance Measurement in Anaphor Resolution 131

5.1. Homonym and Synonym Relation 148
5.2. Ontology example 151
5.3. The four phases of the first spreading activation iteration 155
5.4. The four phases of the second spreading activation iteration 161
5.5. Example for the computation of R.p 167

6.1. Pattern Metamodel 179
6.2. A Pattern, which requires 2..5 persons which sit in a car 182
6.3. One interpretation matching the Pattern 'Persons Sit In Car' 182
6.4. A Pattern, which requires one specific car to have two colors; The cardinality values are values for the minimum and maximum target cardinality attributes 183
6.5. One interpretation matching the Pattern 'Two Colored Car' 183
6.6. An example for a Pattern, where one relations necessity is set to 'Eventually' 184
6.7. Two interpretations matching the 'Drive and Listening Pattern' 184
6.8. Ontology excerpt for the extended matching example 185
6.9. Four similar interpretations for the 'ExtendedMatching' attribute 186
6.10. Two Patterns for the 'ExtendedMatching' attribute 186
6.11. A generic Pattern with an undirected transitivity PatternRelationship 187
6.12. Two interpretations for the 'No Association' Pattern 187
6.13. Overview of Phase 2 192
6.14. Example of states 193
6.15. Example of a Pattern structure with the corresponding states 193

7.1. Framework architecture 203
7.2. Class diagram of the core classes of the Analysis and Pattern components 204
7.3. Part of text 16 which shows the selected words which transport semantically relevant information 216
7.4. Small excerpt of the SemanticScope which has been created for the first case study 216
7.5. Excerpt of the syntactic categories generalization hierarchy 218
7.6. Syntax tree of sentence 28 219
7.7. Part of text 16 which shows the selected words and the structure the Constructions creates 226
7.8. Syntax tree of the first part of text 15 226
7.9. Syntax tree of sentence 17 228
7.10. Excerpt of a UML metamodel for a project management software 234
7.11. Simplified excerpt of the Semantic Scope for case study 2 234
7.12. Simplified InterpretationModel for the command 'Give me all mails about SDSNL', representing only the ConstructionInterpretation elements 238
7.13. Overview of the Pattern mechanism in the second case study 239
7.14. First step of creating a Pattern is selecting the correct elements 241
7.15. Second step of creating a Pattern is adding the relations which represent the required structure within the InterpretationModel 241
7.16. Third step of creating a Pattern is setting the required attributes 242

7.17. Simplified InterpretationModel for command 11: 'Create a note in the wiki of project SEDSNL' 245
7.18. Structure of Pattern 50 245
7.19. Structure of Pattern 51 246
7.20. Structure of Pattern 52 246
7.21. Simplified InterpretationModel for command 28: 'Give me all mails about SEDSNL' 247
7.22. Simplified InterpretationModel for Pattern 170 247
7.23. Simplified InterpretationModel for Pattern 190 247

List of Tables

2.1. OWL Sublanguages [2] 45

3.1. OWL to SE-DSNL Transformation 82
3.2. Textual Representation of Construction *c1* 'University' 84
3.3. Textual Representation of Construction *c4* 'UNA' 85
3.4. Textual Representation Construction NP VP NP 86

4.1. Comparison of concepts related to SE-DSNL 145

7.1. Scenario 1: Updating lexical information 206
7.2. Scenario 2: Editing semantic information 206
7.3. Scenario 3: Modifying Functions 207
7.4. Scenario 4: Updating Constructions 208
7.5. Scenario 5: Changing the syntax parser 209
7.6. Scenario 6: Adding a new Language 210
7.7. Scenario 7: Introducing the SE-DSNL concept to a new but known domain 210
7.8. Scenario 8: Introducing the SE-DSNL concept to a new and unknown domain 211
7.9. Scenario 9: Portability 212
7.10. Construction instances depending on varying numbers of Constructions . 213
7.11. Construction instances depending on varying numbers of Construction-Symbols 213
7.12. Textual Representation Construction APPR + NN + ART + NN 220
7.13. Textual Representation Construction PP + V + ThingRef + ThingRef + PP . 222
7.14. Results of the first case study 224
7.15. Textual Representation of Construction V + NP + PP 237
7.16. Results of the second case study 243
7.17. Commands which have been matched to more than one Pattern in the first phase of the Pattern evaluation 244

A.1. POS tagset of the Penn Treebank [228] 256
A.2. Syntactic tagset of the Penn Treebank 256
A.3. All test sentences and sentence pairs for case study 1 257

A.4. Commands for a fictitious project management software; The row 'Pattern ID' defines the pattern which should identify this command 258

List of Algorithms

1. Mapping Phase Algorithm 105
2. Construction Instantiation 110
3. Fragment Identification Algorithm 117
4. Connections Collection Algorithm 118
5. Solution Path Creation Algorithm 119
6. Fill Solution Paths Algorithm 119
7. Interpretation Models Creation Algorithm 120
8. Syntax Tree based Distance Measurement Algorithm 132
9. Count Nodes to LCA Algorithm 133
10. Syntax Tree based Distance Measurement Algorithm 133

11. Initialization 156
12. Process Tokens 157

13. Pattern Selection Algorithm 189
14. Pattern Rating 191

Curriculum Vitae

Personal Information

Name	Wolf Fischer
Birthday	March 4th, 1982 in Bonn, Germany

Professional Experience

October 2007 to May 2013	**Research Associate** Software Methodologies for Distributed Systems, University of Augsburg, Germany
November 2005 to August 2007	**Research Assistant** Software Methodologies for Distributed Systems, University of Augsburg, Germany
November 2005 to August 2007	**Freelancer** em-tec GmbH, Finning, Germany

Education

October 2002 to September 2007	**Applied Computer Science with an Economics Minor (Diploma)** University of Augsburg, Germany
August 2001 to May 2002	**Civilian Service** Klinikum Landsberg, Germany
September 1992 to June 2001	**Secondary School** Rhabanus Maurus Gymnasium, St. Ottilien, Germany
September 1988 to July 1992	**Primary School** Grundschule Windach, Germany