

Knowledge based Annotation of Requirements and Reuse of Components



Wolf Fischer

TR 2007–10

(Basierend auf der Diplomarbeit von Wolf Fischer)

Universität Augsburg
Institut für Informatik

Oktober 2007

Abstract

Software engineering nowadays is most often a very complex, inefficient, not clearly schedulable and time consuming task. Most of these facts can be related to the software methods used. These have rarely been upscaled to the needs of modern software development. Although reuse has often been called one key in the solution to those problems no methods for an automatic detection of specific artefacts exist. The development of an automatic method which identifies suitable components based on specific requirements would greatly facilitate the requirements- / software engineer's life. Yet, the standard way is to use Google or any other search engine to find matching components.

This work describes an approach to search automatically for already existing components. One major requirement behind this goal is that the user doesn't have to incorporate a lot of additional work as he would have to in other approaches. This means that he simply enters the normal requirement text into the software, which, ideally, automatically analyzes the text and recognizes all kinds of information needed. These would then be marked with tags. In a next step, the so annotated text is inserted into an ontology, which is then compared to the ontology of one other artefact at a time. In the end, a similarity and satisfaction value are returned. Both of them should help the user to find matching components more efficiently than it would take him if he did the search manually. This should reduce the amount of time needed to finish the project as also improve the overall quality.

Starting with a basic introduction about semantic software engineering, the description of the algorithm and the architecture of the prototype is shown. It is then evaluated on different parameter settings, using different example ontologies.

Contents

1	Introduction	1
1.1	Task formulation	2
1.2	Solution idea	4
1.3	Overview of thesis structure	6
2	Technological Background	7
2.1	Clarification of technical terms	7
2.1.1	Ontology	7
2.1.2	Inferencing / Reasoning	8
2.1.3	Requirement	8
2.1.4	Requirements Engineering	8
2.1.5	Artefact and Component	10
2.1.6	Pattern	10
2.2	SemSE	11
2.3	SemIDE	12
3	Semantic Requirements Engineering	15
3.1	SemRE	15
3.1.1	Software reuse	16
3.1.2	Scenarios	17
3.1.3	Integration in SemIDE	22
3.2	Description of an artefact	24
3.2.1	Necessary information and tags	24
3.2.2	Semi-automatic annotation using NLP	25
3.2.3	Basic ontology structure	28
3.2.4	Similarity ontology structure	30
3.2.5	Text to ontology transfer	31
3.3	Artefact comparison algorithm	32
3.3.1	Preparation	32
3.3.2	Phase 1 - Non-functional Requirement matching	32
3.3.3	Phase 2 - Syntactic / Semantic matching	33

3.3.4	Phase 3 - Structural matching	36
3.3.5	Phase 4 - Requirement / artefact similarity computation	39
3.4	Additional algorithms	39
3.4.1	Graph based similarity measure	40
3.4.2	Similarity measure based on Wordnet	40
3.4.3	String based similarity measure	42
3.4.4	Distance measurement in an ontology	43
4	Realization	45
4.1	Requirements	45
4.2	Used software components	47
4.2.1	General	47
4.2.2	NLP	48
4.2.3	Reasoning	50
4.2.4	Similarity	51
4.3	Static Design	51
4.3.1	System overview	52
4.3.2	GUI	52
4.3.3	Inbuilts	54
4.3.4	XML	56
4.3.5	Design Patterns	56
4.4	Dynamic design	58
4.4.1	Basic relevant functionality	58
4.4.2	NLP functionality	62
4.4.3	Similarity functionality	63
4.5	Implementation	63
4.5.1	Jena and its rules	63
4.5.2	Concrete tags and file structure	64
4.5.3	Graphical User Interface	65
5	Evaluation	69
5.1	Evaluation criteria	69
5.1.1	Parameters	69
5.1.2	Comparison	71
5.2	Basic parameter values	71
5.2.1	Syntactic / semantic parameter evaluation	72
5.2.2	Structural parameter evaluation	73
5.2.3	Requirement parameter evaluation	78
5.3	Requirement document evaluation	80
5.3.1	Testdata	80

6	Summary	85
6.1	Related work	85
6.1.1	Semi-automatic Semantic Annotation	85
6.1.2	CARE Approach	85
6.1.3	Other related work	86
6.2	Outlook	87
6.3	Conclusion	88
7	Appendix	97
7.1	Usage instruction for the GUI	97
7.2	Scenarios	98
7.2.1	Nowadays - Scenarios	98
7.2.2	Nowadays - Component search	100
7.2.3	Intended scenarios	100

Chapter 1

Introduction

Today's life is unimaginable without computers. No matter which job or which area a human works in - everywhere he will be supported by computer systems and the software they are running. Software supports schools in planning their timetables, banks in doing their financial transactions, drivers while steering a car etc.. Due to the evolution of technology and the higher requirements of companies, the complexity of software also increases. This often leads to delayed or even cancelled software projects. An additional problem is the enhancement of already existing software. Perhaps it has been developed decades ago and evolved over time. Because of old methodologies used (or none at all), it may be hard to keep track of - there are no easily separable parts identifiable, no documentation models exist etc.. Another possibility would be that the software has already become so huge in size that understanding parts or especially the whole of the project is a nearly impossible task, although documentation exists. 'Impossible' in this case mainly refers to a very time consuming and therefore money-intensive undertaking. One way to solve this has often been propagated, but never really been adapted in practice: The reuse of components. Regarding difficult tasks, reuse of course happens (no company would write a complete database on its own), but especially on smaller jobs, this is not often the case (e.g. during development one team member had written a method, which he did not tell anybody about, so someone else writes a method with the same semantics).

Certain approaches already exist to shift the focus onto components. This could be either a whole methodology, focusing on components and/or ways to search for them. These search engines most often have two different ways of annotating the component description:

- The components are described by using special formalisms.
- The normal textual description of components can be searched by using keywords.

Formalized annotations have proven to be capable of solving the problem per se. Still, annotating components with a formalized language is most often a difficult,

time-consuming and at least a learning-intensive undertaking for humans. The problem with keyword powered search engines is that you either get several dozens or even hundreds of results, which simply contain the keywords, or you get no results at all. Even in case that a set of results has been returned, you have to find the fitting components in the result set yourself. Additionally you cannot be sure that the results really contain what you were looking for, because you used a synonym or a semantically closely related term.

A possible solution to this problem will be described in this work: A (semi-) automatic artefact search. The goal is to give either the requirement- or the software engineer a tool which takes the requirement text and uses it to search for similar artefacts. These may either to be implemented or already existing. The term 'semi-' refers to the problem that user input is still required. The reason for this is that certain pieces of information can not yet automatically (i.e. through the use of computers) be extracted from natural language texts. This way, a new software engineer could find certain artefacts in an existing software project without having to ask colleagues or to debug the system for several hours, days or weeks. A requirement engineer in turn could automatically see which components already match his requirements. This in turn could lead to lower costs, increased development time and enhanced quality due to the reuse of existing components. These could additionally be directly incorporated into the design of the architecture and this way further increase the overall quality of the project.

1.1 Task formulation

Before specifying the definitive task of this thesis the problems to be solved will be explained in greater detail.

1. Basic artefact descriptions often only consist of a text written in natural language. The problem with this is that it is difficult for computers to analyze. Information extraction or the recognition of certain type of words is a huge field of science, which retrieves more and more promising results. Still, extracting necessary information from text corpora is more easily and accurately for humans to do. However their problem lie in the amount of time they have to put into this process. Also, repeating one and the same process over and over again results in quality problems due to monotony. This is a definitive strength of computers, which can repeat the same process on large sets without a loss of quality.
2. Searching for suitable components is, as already mentioned, today most often done using a normal search engine. It uses some keywords and lists all web pages which contain the corresponding words. The gathered results contain items which

are not about components at all or do not describe any relevant artefacts. This may be due to wrong technical terms. It is therefore a very time-consuming undertaking to find matching components.

3. If there are search engines specialised for the detection of components, they also often simply use keyword based mechanisms. In case they do not, logical formalisms are used to describe the artefacts. The advantage of this approach is a very accurate description. Disadvantages are the amount of time which is needed to describe the components as well as the lack of expressiveness and simplicity.
4. Mostly component reuse is only propagated using COTS components. However large projects also tend to consist of hundreds of methods and classes. All of them are capable of solving specific problems and are therefore seen as an artefact. Because of that it should be possible to search these items, too. In general, it should facilitate the orientation in projects as well as also help reducing the overall redundancy.

The task of this thesis is to develop a concept which demonstrates that it is per se possible to solve these problems above. However, it is not necessary for the results of this thesis to provide a fully operational concept as well as software prototype. More over a general theoretical concept as well as a prototypical implementation of it should be made. The concept should show a way how existing artefact descriptions can be used to search for other similar descriptions. More detailed, the following requirements must be satisfied by the concept:

1. If an existing informal description can not be used easily, the user must be able to add the necessary information as fast as possible.
2. Generally, the user should not have to put a lot of effort into using the prototype.
3. Components must be found based on their description.
4. Not only COTS components but also artefacts like methods or classes should be found.

Showing a way how these tasks can be solved and realizing the solution should facilitate the life of a software or requirement engineer. Ideally the user should only enter its descriptions and the software automatically determines which artefacts might be suitable. Based on the idea of semantic software engineering, an idea for a possible integration into the SemIDE - a vision, created by Bauer and Roser ([32]) - must be shown.

To demonstrate the viability of the concept, a prototype must be implemented. Its core functionality should be for one the creation of an artefact description as well as

the component matching process. It is not necessary for the prototype to implement a fully working repository. Also it must not be optimized for high efficiency or the best possible user interaction.

Finally the prototype should be evaluated based on a sample data set. Based on the results gathered during the evaluation the overall realizability of the concept can be estimated. Independent of the evaluation results, potential future enhancements of the concept and the prototype should be shown.

1.2 Solution idea

Regarding the problem of using the description of a component the question was which kind of description to use. Logical formalisms seem to contradict the tasks which have to be solved in this thesis. Using simple normal textual descriptions and compare them on a keyword basis is not detailed enough. However, it would have the advantage of a relatively small effort a user would have to muster. The solution proposed in this thesis lies somewhere in between. Because one part of the task is to not burden the user with additional effort the idea is to take the normal natural language text, annotate specific information and use it for the comparison process. The question is which parts of natural language texts are of relevance and how different texts could be compared. A normal text and the order of its words can be seen as a graph. First of all, normal text forms a chain where each word is connected to every word that follows him in the text corpus. In this chain some words also have relations to other ones. An example is given in figure 1.1. Basically, the graph in this case consists of two sentences: "Tom walks. He is busy". The dot in the sentence is of no further relevance and is therefore left out in the graph. What is difficult to detect is, which words refer to the same entities. In this case it can easily be seen that "Tom" and "He" mean the same real world object (indicated by the orange arrow). Based on this fact it can be reasoned that Tom is also busy, i.e. Tom can also be referred to the word "is" (indicated by the green arrow). As already mentioned in section 1.1, information extraction and especially the detection of so-called coreferences is a difficult task for computers. Still it could facilitate the annotation process, supported by the user, by finding at least some of all existing coreference relations. The rest of all necessary information must be annotated by the user.

Annotating normal text is per se an easy process if it is tool driven. It can be compared to changing the color of a text in a word processor, i.e. the user marks the corresponding part of the text and clicks the "Change Color" button. Further, the annotation of normal text can be easily incorporated into the documentation of existing source code. This would help satisfying the requirement that every kind of artefact must be detectable. JavaDoc already possesses a good foundation which could in this case be enriched with additional tags.

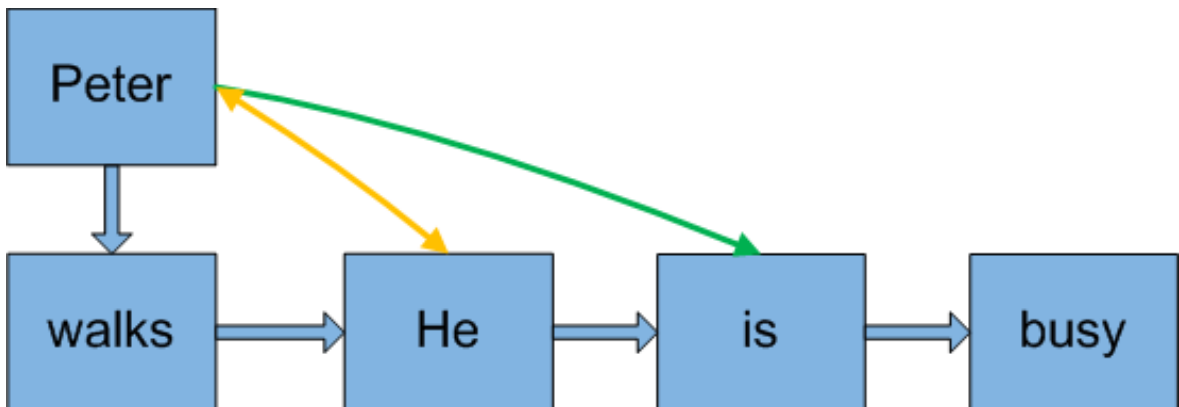


Figure 1.1: A simple text chain

Not only coreferences are of importance to the matching process but also the separation of requirements into normal and non-functional (NF) ones. The latter have a special priority as components which do not have matching NF requirements must not be considered any further. Also, it could be possible to extract at least certain NF requirements automatically. This is especially the case if they describe for example system information like "Windows" - a namefinding component, for example executed during the NLP mechanism, could detect this information and annotate it without user intervention.

Identifying the similarities between graphs can in this case be done with a string as well as a semantic based comparison mechanism. The idea is, to compare every word of one graph to the word of another one. The string based comparison should detect all equal (and perhaps mistyped) words. The semantic comparison, instead, should identify all semantic close terms like "auto" and "car", using a machine-readable "dictionary". After that a contextual similarity matching is necessary to ensure that both graphs describe a similar kind of artefact and do not only have similar words in their text corpora. Based on these pieces of information an overall similarity value can be calculated for every requirement. The similar requirements will then be used to generate a final value for the similarity between two artefacts.

One question which has not yet been mentioned is which technologies should be used to represent the graph as well as perform the necessary calculations on the graph itself. Looking at the Semantic Web, standards like OWL have been developed which exactly support the idea above. Ontologies can be seen as graphs which are therefore suitable for the containment of text. Additional facts are then generated on top of all existing information using the help of reasoners and a corresponding rule set, which represents the logics needed for this thesis.

1.3 Overview of thesis structure

This work is structured the following way: This chapter gives a short introduction on the subject. Chapter 2 describes the technological background in detail. Therefore, technical terms will be explained as well as the idea behind semantic software engineering is. In chapter 3, the big picture of SemRE (Semantic Requirements Engineering) is explained and what the tool to be developed should do. Additionally, a possible integration into SemIDE (Semantic IDE) is shown. Next, the algorithm for artefact comparison is laid down in detail. Therefore, it explains, what the description of a component looks like and how the corresponding ontology structure works. After that the different phases of the artefact comparison algorithm will be expounded, before the algorithms for the similarity matching are presented.

Chapter 4 then describes the implementation of the theoretical foundation. It starts with a description of the requirements, which must be fulfilled by the prototype. This is followed by a list of components, which have been selected for implementation, before the static structure of the architecture is introduced. Finally, the implementation of the algorithm will be shown. In chapter 5 the prototype will be evaluated by using different examples and parameter settings. Finally, chapter 6 delimits this work to already known / existing / similar approaches. Next, it gives an outlook on the possibilities for enhancements, what future steps could look like etc., before the whole work will be summarized in the end.

Chapter 2

Technological Background

The idea to SemRE came from two different papers ([1], [34]). Both propose the idea of using semantics for software engineering. Most ideas presented in these articles have not yet been worked out. Basically this thesis tries to show that one part of these papers could be realized by using today's semantic web technologies.

Both articles propose the idea of using semantics for a higher industrialization of software to have a) a higher product quality, b) faster development life cycles and c) a better competitive standing against low-wage countries. What a higher degree of automatization can offer an industry has already been seen in the automotive and other production oriented domains. First steps into the software industrialization direction have been taken with OMG's MDA ([29]) initiative by using automated model transformation and code generation. Still, there is much more room for enhancements which would need the incorporation of semantics.

This chapter first gives an explanation of different technical terms which are of relevance for the rest of this thesis. Next, the basic idea behind semantic software engineering (SemSE) will be described, before the vision of the SemIDE is being introduced.

2.1 Clarification of technical terms

To allow a better understanding of the rest of this work, several technical terms will be described in more detail.

2.1.1 Ontology

The definition for ontology is taken directly from Gruber ([11]):

An ontology has been defined as a (formal) explicit specification of a (shared) conceptualization. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts

of that phenomenon. Explicit means that the type of concepts used and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group.

An ontology contains so-called ObjectProperties, which will later on be referred to as relation, i.e. simple relation between two different classes. Further, so-called DataTypeProperties will be referred to as attributes of a specific class.

2.1.2 Inferencing / Reasoning

Based on the definition in [32], inferencing describes the additional generation of facts / information. These facts are generated from an already existing pool of information. The terms 'reasoning' and 'inferencing' will be used synonymous in this paper.

2.1.3 Requirement

Based on the definition of [33], a requirement is not a uniform term. In some cases a requirement can be seen as either an abstract, more general description of a service, which the system has to achieve, or a restriction of the system. On the other hand, a requirement is a detailed mathematical, formal definition of a system service. The existence of these differences is based on the fact that in the first case a solution should not be anticipated. As such, several competitors are able to differentiate themselves from the competition with different solution approaches. The more detailed, mathematical definition is called a system definition and is created by the contractor, after the contract between the client and the contractor has been signed. Thus the client can understand and gauge on what the software will do.

In the case of this thesis, a requirement is seen as an abstract and more general description of a service.

2.1.4 Requirements Engineering

Requirements engineering describes the process of identifying the intended purpose of a software (Nuseibeh and Easterbrook, [28]). One detailed definition is given by Zave ([39]):

Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families.

The process is a very complicated one, because stakeholders of the new system may be numerous and distributed as well as have different goals, which themselves may conflict. Also the description of their goals may not be explicit enough etc..

The whole RE process is split into different activities, which are presented in the following. Most of them are supported by different tools, which are offered on today's software market.

Elicitation

The elicitation of requirements is often seen as the first step in the whole RE process. It tries to identify the system boundaries, which define, at a high level, where the final system will fit into the current operational environment. Next, relevant stakeholders must be identified. These are persons who are somehow influenced by the new system. Finally, the goals of the new system must be gathered. This focuses the requirement engineers on the problem domain. Possible techniques for eliciting requirements are for example surveys, given to the stakeholders or brainstorming sessions. In cases of great uncertainty, prototyping may also be a reasonable choice.

Modelling and analysing requirements

Modelling describes the construction of an abstract description. The analysing process then gathers further information from these models. Many different categories of RE modelling approaches exist. Some of them are:

- Enterprise Modelling - Most RE activities and software systems take place in an organisation, in which the development of the new system takes place. Enterprise modelling and analysis tries to describe the organisation's structure, its business rules, goals etc.
- Data Modelling - Computer-based systems often produce large amounts of information, which needs to be understood. Traditionally, Entity-Relationship-Attribute or object-oriented modelling are just some of many different techniques, used for modelling and analysis.
- Behavioural Modelling - Trying to specify the dynamic and functional behaviour between a system and its stakeholders is a very important key in Requirements Engineering. Possible methods range in this case from structured to object-oriented methods as well as from soft to formal methods.

Communicating requirements

Communicating the discovered and specified requirements to different stakeholders is a very important task. The documentation of requirements ensures that they can

be read, analyzed and validated. Many different ways exist to describe requirements, ranging from formal to informal languages. They differ in their expressiveness as well as their reasoning capabilities. Another very important area is the field of requirement management, which tries to specify requirements in not only a readable, but also by many users traceable form. This should help managing a requirements evolution over time, i.e. forward (from its origin through the development to its subsequent deployment) as well as backwards.

Agreeing requirements

After requirements have been collected as well as models been created and analyzed, stakeholdes must agree on what has been gathered, which is especially difficult with divergent goals. Validating that requirements and models provide an accurate account of stakeholdes requirements is a very complex task. Depending on the documentation structure different techniques (e.g. SCR ([14])) can be used to validate at least certain parts of requirements, for example if they are consistent.

Evolving requirements

Systems operate in ever changing environments. Therefore their requirements have to evolve over time. Managing change is a fundamental RE activity. It requires the possibility of configuration management and version control as well as the monitoring of traceability links.

2.1.5 Artefact and Component

In general any object crafted by a human being can be seen as an artefact. In this paper an artefact is anything from a simple method to a class to any finished software product, even COTS (Commercial-Of-The-Shelf) components. To end confusing the reader, artefact and component are being used synonymous in this thesis.

2.1.6 Pattern

Many definitions exist about what a pattern is and what it stands for. One of the most common ones is (see [5], [9])

A pattern is a proven solution to a problem in a context.

However, this definition only comprises a small and very abstract part of what a pattern is and what it can really do. Some see a pattern as "a way to chunk up advice about a topic" ([9]). This way, a huge amount of knowledge can be divided so that it can be remembered more easily, if necessary. Another definition is given by Kurotsuchi ([19]):

Patterns are devices that allow programs to share knowledge about their design.

As can be seen from these two examples it is very difficult to exactly specify what a pattern is. Therefore, both given definitions are just two out of many different ones which can be regarded as true.

2.2 SemSE

Kurtev ([20]) showed a way, how different technological spaces (TS) could be joined in an effort to have a higher benefit from the different TS. Examples for different TS would be the MDA TS, XML TS, DBMS TS, abstract syntax TS and so on. The ontological TS deals mainly with representation and reasoning. It has great advantages in the field of traceability (specification of correspondences between various meta-models). During the last years and the evolution of the semantic web, matured reasoners have been developed which enable good deduction systems, consistency checking, validation of models etc. which, in turn, greatly support automated software development. There are many fields in the software engineering area which can be enhanced with semantics:

- **Methodologies** - A methodology consists of a modelling language and a software process. The latter has different activities which result in one or more deliverables, e.g. documents, code, reports and especially models. Transforming the model of one activity into the one of another activity is a difficult task, which could be facilitated, using semantics. Further, it could lead to an automatic document generation during the development process or a better adherence of custom modelling guidelines.
- **Interoperability** - The need for exchanging information between different corporations is a well known problem. Ontologies could help exchange models and describe their semantics, independent of the modelling style of the different corporations. Additionally, ontologies can be used to compare meta-models and automatically derive mappings from one meta-model to another.
- **Web Service management** - The Web Service field contains many standards, most of them belonging to the WS*. Although most of them are complementary, they overlap in certain field. One of them may even need to produce some models, which are composed of other WS standards but may be inconsistent in certain areas. A coherent semantic description could help to reveal these inconsistencies.
- **Feature model verification** - Feature models are an important role in software reuse. They represent distinctive user visible characteristics of a system.

However, the lack of formal descriptions has restricted the further development in this area, although this would be of great use to the industry.

- **Search and composition of components** - A semantic description of components and services could be stored in a knowledge base where ontological reasoning and matchmaking mechanisms could be used to find similar components. Also these API descriptions can support an automatic consistency check or the search for necessary components.

These are just five of many areas, which could be enhanced with semantics. The last one, however, is the origin of the idea to this thesis.

2.3 SemIDE

SemIDE is the acronym for semantic-enabled Integrated Development Environment. It helps the user to create models of the MDA TS and enhance these with ontologies from the ontological TS. Models of the Ontology TS are being represented by application ontologies and synchronized to the MDA TS via so called bridges (these work on the abstract syntax of both TS). To further enhance the reasoning capabilities of the Ontology TS and to obtain unambiguous semantics of the models the application ontologies are bound to reference ontologies.

As can be seen in figure 2.1, meta-models are being added to the SemIDE (1) before modelling starts. Their corresponding application ontologies and bridges (3) can be generated automatically with concepts described in [30]. Application ontologies, bridges and bindings which concern models are added to the SemIDE during modelling time (see (2) and (4)). Reference ontologies (5) can be deployed before and during the modelling time.

The architecture of the SemIDE can be seen in figure 2.2 and basically consists of an infrastructure as well as an extension mechanism which allows the plug-in of multiple so-called Sem-X-Tools. The infrastructure holds the basic functionality of the SemIDE like modelling tools, model transformation engines, reasoning components as well as the application ontologies and model repositories. It therefore also realizes the bridges between the models and application ontologies.. Further, the infrastructure offers interfaces which allow the Sem-X-Tools to access the features of those components. The methodology repository is for one part of the infrastructure, but can also make use of the Sem-X-Tools. Therefore it checks which Sem-X-Tools have registered at the infrastructure to obtain information about these.

Sem-X-Tools use the SemIDE infrastructure to enhance the overall capabilities of the whole SemIDE. They consist of a **model manipulator** (reads, creates, modifies models of the model repository (1)), a **Sem-X-Component** (which implements the logics of the Sem-X-Tool, gets information about models through the model manipulator (2),

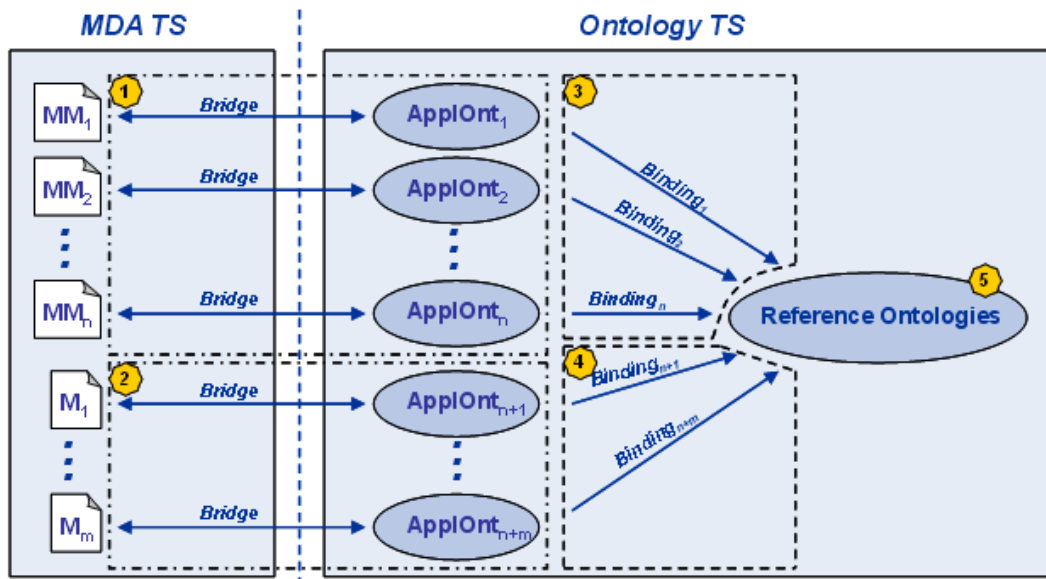


Figure 2.1: Modelling in SemIDE

provides interfaces for model manipulation itself (3) and computes model adjustment operators (4) based on the results, gathered by the reasoning mechanisms) and a set of reasoning **rules** (which are loaded by the reasoner (5) to infer additional facts). These three elements are unique for every Sem-X-Tool.

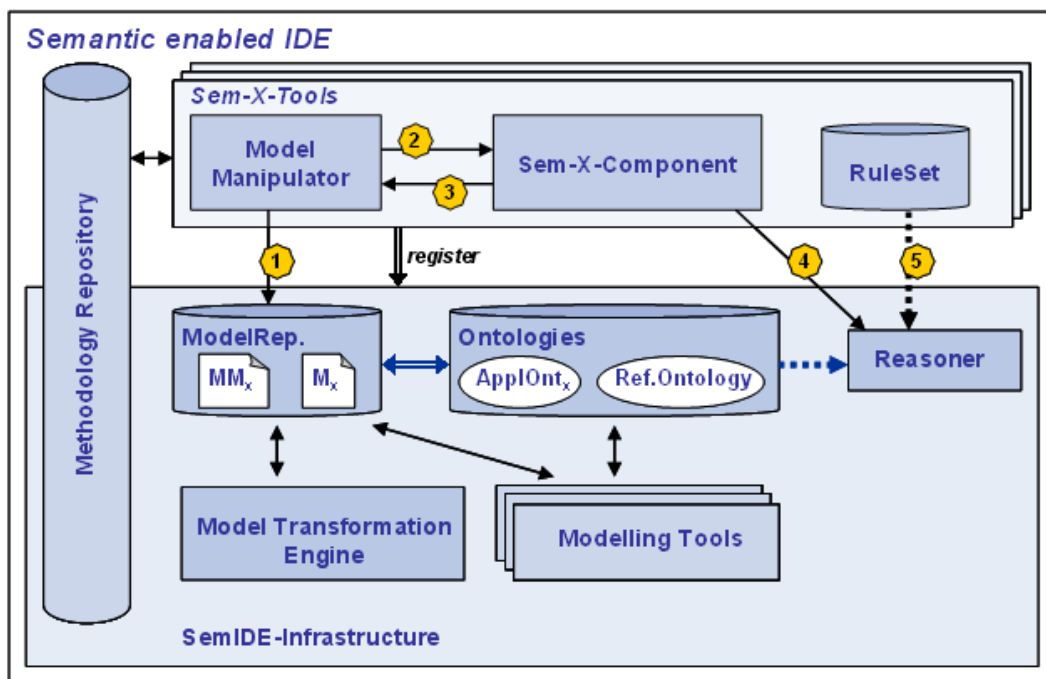


Figure 2.2: Possible architecture of SemIDE

Chapter 3

Semantic Requirements Engineering

The comparison of two artefacts is based on the comparison of their descriptions. The idea of this diploma thesis is to relieve the user of as much work as possible, i.e. compared to other approaches (see 6.1) the user should not need to learn special and probably complicated syntactical formalism, but simply enter the textual description and annotate the text with certain tags (which can happen semi-automatically). The rest can be done automatically with the algorithm, described in more detail in the following sections. Therefore, four phases have been developed. The first one searches for matching NF-requirements. If this phase succeeds (i.e. there are enough NF-requirements) the second phase begins. It tries to identify as many similar word tuples from both descriptions as possible. Based on these results, the third phase computes a structural matching. It detects a context related similarity. Finally, the last step generates a similarity value for both the requirements as well as the artefacts themselves.

The structure of this chapter is as follows: First, a general introduction into semantic requirements engineering is given. Next, the description of an artefact as well as its tags, annotation and ontology structure will be shown (see 3.2). In the following, the idea behind the artefact comparison algorithm will be explained (3.3) before, finally, additional supporting algorithms which are of relevance to the concept will be presented (3.4).

3.1 SemRE

Requirements (see 2.1.3 for a precise definition) are usually one of the first pieces in a development process and therefore have an impact on everything which follows in a later development phase. The description of requirements, often being done in natural language, results in many different problems. These are:

- Fusion of requirements - Sometimes, different requirements will be seen as one requirement, although they are not.
- Overlap of requirements - Several requirements may describe the same feature, but will be seen as different ones.
- Lack of accuracy - Bringing natural language into a precise and unmistakable form is a difficult task to do without making the document huge and difficult to understand.
- Contradictions - Especially on large documents it may happen that parts of one requirement contradict some or even all statements of another one. Finding these contradictions can reduce the overall risk of a project delay or even failure.

These are problems, which not only have an impact on the requirements engineering process itself, but on the entire project development. Detecting problems of that kind in an early phase can prevent nearly complete projects from failing or at least from being delayed for a certain period of time because the later a failure is being detected, the more expensive it is to repair.

As already mentioned, requirements are the golden thread which guides the rest of the software development process. As such, every person, involved in the project, will read the document most probably written in partially ambiguous language. Therefore, there is a high potential of additional problems arising during the development itself just because of different people understanding the requirements in different ways.

A solution to these problems could be a semantically enhanced requirements engineering (SemRE) process. This imposes the idea of semantically described requirements, supporting the development of any software artefact. The new information, gathered through these semantic requirements in the very beginning of a project, can then be used to guide the software engineer through the rest of the whole development cycle. Using the semantic descriptions of the requirements, inconsistencies and incompleteness as well as traceability links between artefacts and their requirements (see [40] for a possible approach) could be detected at an early stage of the development process, using reasoning mechanisms (see [16]). One further use of semantically annotated requirements is to automatically find matching artefacts at the beginning of a development process, which could lead to a better integration into the software which is to be developed and, more generally, a reduced amount of developing time (therefore a shorter time to market) as well as an enhanced quality (performance, stability, security, etc.) of the final software.

3.1.1 Software reuse

Because of the ever increasing size and complexity of software projects, one paradigm which has been proposed for years is the reuse of software, especially of components.

This method is already common in every other production based industrialization area like automotive (where the same component for ABS is used by different car manufacturers) or even computer assembling (the same processor type is used by e.g. Dell and HP for their laptops): Every company shifts its focus to one (or more) key areas and develops a product, which does exactly what it is meant to do and should therefore have a high quality (because of just focusing on special features). Still, this paradigm does not have the same impact on software as it had on the hardware area. Reasons for this may be for example of either legal (is the software distributed, using the correct license) or financial kind. Regarding the last one, it could either be the case that the components are too expensive for what is really needed or the person who has to evaluate a certain kind of program type and all its available competitors simply needs a huge amount of time ("Time is money"). Perhaps this person is not even an expert and therefore does not know what the correct terms for searching his components are. Therefore, it could very well be the case that he does not find potentially matching components because of the use of synonyms (this fact can not be handled by a search engine). Another possibility would be that the components he needs are listed on page 150 out of several thousands, depending on the number of results the search engine returned for the key words the user entered into the search engine mask.

3.1.2 Scenarios

So far, only the problem to be solved has been shown in greater detail. The exact way, how the SemRE tool could work, how it could be incorporated into today's project work and which changes have to be made to an existing methodology to gain advantages of the SemRE tool, have been left out.

To gather more information about how the tool could be used some effort has been put into detecting what exactly the possible scenarios for the SemRE tool are like and how it could help in the development of a software. Generally speaking, the overall idea of the SemRE tool is a fast and easy input of requirements and their annotation. This should be independent from the case in which the requirements are entered, i.e. it should not matter, if a requirement engineer at the very beginning of a project enters its requirements, if a software architect takes the requirement document and imports it into the SemRE tool or if a programmer wants to describe a single method or class, which he implemented - All the time, he must have an easy and fast access to every possible annotation function.

In the following, today's as well as future's scenarios are shown, described and the intended advantages through the use of SemRE presented. However, the scenarios, following, are not complete. The rest can be found in the appendix ([7.2](#)).

Scenario 1 - Today

The first case in table 3.1 describes a scenario as it can very well be the case today (note: The ideal case of a professional requirement engineer analyzing the circumstances and requirements for a software tool is not always standard.) A software engineer has got a requirement document for a software which is to be developed. Therefore he just starts right away with the development. It would have been of help to the client, if he could have seen an overview of existing components, which could accelerate the whole engineering process. This way, the software engineer might develop every single part of the software on his own (which is most often even financially more rewarding than using existing components).

ID	1
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming has not started yet - Semantically ambiguous requirements
Action	Does not care for probably existing components (time / motivation / political reasons) and simply starts developing
Result	<ul style="list-style-type: none"> - Pure / less quality than when using existing components - More time needed for development of software

Table 3.1: Scenario 1 - Today

Scenario 1 - Future

The scenario in table 3.2 probably describes the most basic case. An end-user or the requirement engineer uses the SemRE tool to enter and annotate the requirements he has. With this information, the tool automatically starts searching for matching components. The most suitable ones can then be taken and directly used in the design phase. This offers the advantage of not having to break a design which is possibly already existing. It should therefore be possible to create a more structured architecture. The negative aspect is of course the time which is needed to annotate the requirements. Because of the better architecture and the fact that less time is needed to find the components, this disadvantage should be more than compensated.

Scenario 2 - Today

A completely different case is shown in table 3.3. In this case, a software programmer who is writing code for a big and perhaps even distributed project needs a special feature, which could be solved in a single method or class. Due to the complexity of the project and the difficulty that he just joined the development team, he lacks a lot

ID	2
Role	End user / requirement engineer
Condition	<ul style="list-style-type: none"> - Using SemRE - tool - Wants to describe the features for a new software product
Action	User writes and annotates its requirements in a special editor, which puts all the information directly into an ontology
Result	<ul style="list-style-type: none"> - More time needed for annotating requirements - SemRE tool automatically searches for components, which satisfy some of the requirements. Those components then can be directly incorporated into the design-phase

Table 3.2: Scenario 1 - Future

of knowledge about the architecture of the project. Therefore he uses the framework library or an Internet search engine to find a method which satisfies the feature he wants. Still, this search process can take a lot of time. Perhaps he may find the method, if he continues the search, but the amount of time this takes forces him to write this method on his own. Probably he will find a similar method later (written by someone else), which does the same in a more efficient way.

ID	3
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming has started - Programmer needs a software routine - Semantically unambiguous requirements
Action	Searches the framework library and / or uses an Internet search engine to find a suitable method, but cannot find one (in a certain amount of time). Starts writing the method himself
Result	<ul style="list-style-type: none"> - Self written method with potential bugs due to not efficient testing - Time lost with something that had no results

Table 3.3: Scenario 2 - Today

Scenario 2 - Future

In table 3.4, the SemRE tool is being used by a software engineer. This engineer wants to add a new method which he thinks does already exist, but he does not know where. Therefore, he starts by entering the code documentation of the method which is to be developed. While entering the text, the SemRE tool supports him with the

corresponding tags. After having entered the documentation of the method, the tool takes this information and again searches for a method, which might have a similar semantic. This way, the programmer is relieved from manually searching a big part of the existing code basis. If the method exists, there should be a good chance for the tool to find it, so that the user does not have to write the code once again, which reduces redundancy.

ID	4
Role	Software engineer
Condition	- Using SemRE - tool - Wants to find a method which satisfies certain requirements
Action	User specifies the features in analogy to specifying the requirements for the requirement document. The SemRE tool automatically parses the information, adds it to the ontology and searches for matches.
Result	- More time needed for annotating requirements - SemRE tool automatically searches for a corresponding method in the whole project which satisfies the user's requirements.

Table 3.4: Scenario 2 - Future

Scenario 3 - Today

Table 3.5 shows a case in which only the requirement documents exist. The next step would be to start the design of different models e.g. class- or sequence diagrams. The engineer therefore starts creating these. Sometime after this process has begun components are detected which seem to be suitable to the overall requirements. However, they do not match the already existing models. Therefore, these have to be changed which results in an additional time effort. Also the overall quality of the project may suffer.

Scenario 3 - Future

Using SemRE the scenario from 3.1.2 can be speed up by an unknown factor. Table 3.6 shows the corresponding scenario. The conditions are the same as in 3.1.2. However, the engineer takes the requirements and initiates an automatic search for components. He then can use the most suitable ones and incorporate them directly into the design of the models. This results in a higher quality as well as a smaller amount of time used for the creation of the models.

ID	5
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Next step would be the design of a software - No components have been selected
Action	The engineer start with the creation of the models.
Result	<ul style="list-style-type: none"> - Suitable components have been identified later on. - The already design must be changed which results in a loss of time and probably quality also.

Table 3.5: Scenario 3 - Today

ID	6
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Next step would be the design of a software - No components have been selected
Action	User enters the requirements into SemRE. This presents him possibly matching components. These can directly be incorporated into the models.
Result	<ul style="list-style-type: none"> - More time needed for annotating requirements - Components have been directly incorporated into the design phase which enhances the overall quality. Also the creation of the models does take less time than adding components in a later phase.

Table 3.6: Scenario 3 - Future

Scenario summary

All these previous examples are cases, where a SemRE tool with a fast annotation of requirements and an automatic search for components could present a not to be underestimated help regarding development time, code quality and money. As it can be seen, the SemRE tool does not propose a whole new methodology or a big change in an existing one. It simply supports the user in what he is already doing, but where most of the work is still done by him. This should also help a possible distribution of the software - existing approaches do not have to be changed at all, which saves a lot of time and money. They are simply being enriched by the use of this tool.

3.1.3 Integration in SemIDE

The previous sections showed some of many ways how SemRE could help the daily project development. However, the realisation itself has still been left aside yet. As SemRE is thought to be a Sem-X-Tool which itself uses the SemIDE infrastructure a possible realisation of the SemIDE infrastructure will be shown first.

The SemIDE proposes the idea of having a semantically enhanced IDE which takes the best of both worlds (with and without semantics) to help the user. As the reinvention of the wheel is a difficult task, the market itself already offers many well-working IDEs. As this thesis is all about the reuse of components, an already existing IDE should be taken, which must should be easily extensible in every imaginable way. The probably most famous is, in this case, [Eclipse](#). Eclipse is known for its wide distribution, its continuous development, its huge set of already available plugins and a well defined and documented API. Therefore, it seems to be a good cornerstone for an eventual development of the SemIDE - the already existing components of Eclipse just have to be extended with the help of the ideas, shown in this thesis.

Figure 3.1 shows an overview of a basic high level architecture for the SemRE tool. As it can be seen, SemRE is thought to consist of three different components, all of them being Sem-X-Tools:

- Sem artefact Search - This is the cornerstone for the whole SemRE tool. It manages the insertion of text which has been annotated into the ontology and automatically searches all similar components, using the reasoning engine provided by the SemIDE infrastructure.
- Sem IDE Enhancement - One of the scenarios already pointed out that a programmer should be able to add and search for annotated text inside the source code documentation (e.g. java doc). This tool will enhance the SemIDE Intellisense mechanism (a feature, which helps the user to complete words while typing. This can already be seen in any modern IDE like Eclipse or Visual Studio) with the corresponding and available tags (see 3.2.1 for more information),

so that the user can more easily and faster enter all necessary information.

- Sem Requirements Engineering - The annotation of existing requirement text is a longer undertaking than simply entering some java doc text. Therefore, the requirement engineer must be supported by a tool which helps him to annotate the text as fast as possible. It should contain NLP technics to find as much information as possible without the intervention of the user. After all information has been gathered, it is given to the Sem artefact Search tool, which starts a search for matching artefacts.

These three elements could build the basis of what is called SemRE - semantic requirements engineering. Because of the size of the whole project, only parts of the "Sem artefact Search" and "Sem Requirements Engineering" have been implemented during the work on this thesis. The following section now describes in detail how the similarity matching process will work, before the next chapter presents the architecture of the prototype.

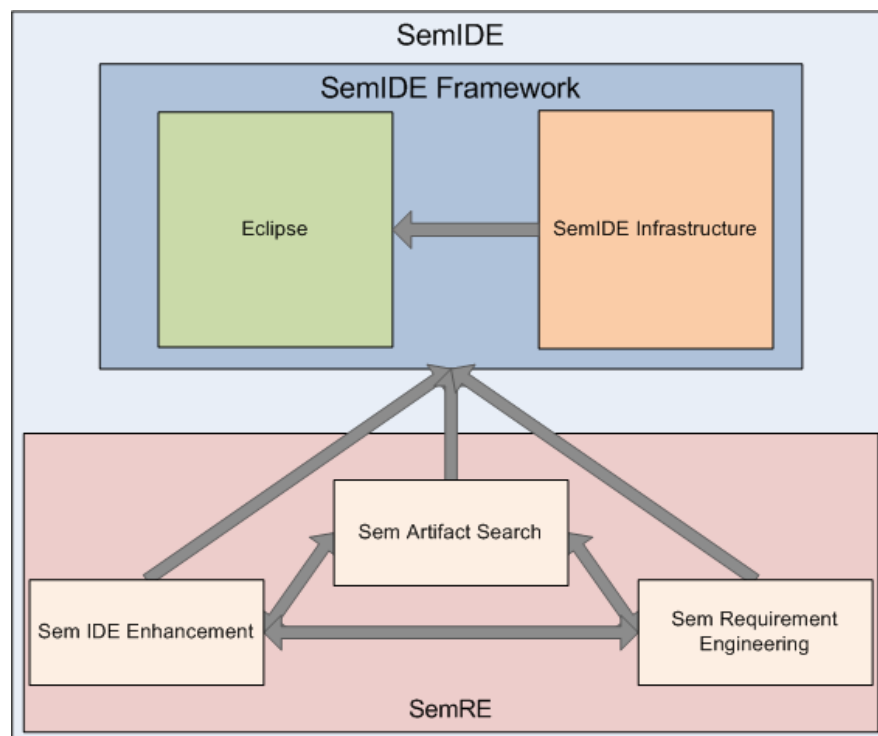


Figure 3.1: Integration in SemIDE

3.2 Description of an artefact

As already mentioned, the user should not need to learn a special syntax and / or semantic to be able to describe an artefact. Furthermore, no matter, whether it is a method, a class or an entire software project, everything can be described easily using natural language. The idea is that everything satisfies one or more requirements. For example, a method could simply convert from one metric system to another: "Converts from kilometers to miles". This is, of course, a very simple example. Projects often tend to have requirement documents which cover several hundreds of pages. Still, they simply contain requirements in textual representation. Additionally, those requirements are most often not just listed in a numerical order, but structured in a tree-like hierarchy. The question is, how this amount of information can be separated into smaller parts which can then be analyzed more easily. This will be shown in the next section.

3.2.1 Necessary information and tags

The previous section already referred to the point of splitting text into smaller pieces, which in turn will be annotated by tags. Before showing the different types of tags, the kinds of information which must be tagged and are therefore of relevance for this work will be explained.

As already mentioned, projects tend to consist of requirement documents, which can be seen as the whole source of information for an artefact. This information is split into its requirements. There are two types of requirements:

- Requirement: A requirement describes certain features, which should be satisfied by a project.
- Non-Functional (NF) requirement: An NF-requirement is a specialization of a normal requirement. It contains all facts which are known to have nothing to do with the functionality of the software itself. This could be, for example, the operating-system, the software which is to be developed should run on or the processor performance the software needs.

The most important information, which is needed for the algorithm to deliver decent results is, which words in the requirement document describe the same facts, i.e. refer to the same entities. This problem is often referred to as coreference-resolution. If looking at the way humans write down information, you will notice that this can be dispersed over the whole document, e.g. a part of a car's engine is described at the very beginning, whereas further details of the engine are explained at the end of the document - i.e. the pieces of information do not have to be next to each other in the text. The problem is that it is extremely difficult for a machine to find out which

words refer to which other words. Even for humans this task often is not that easy. Still it is probably the most essential part of this thesis, because it helps "clustering" the information. This is best demonstrated using a simple example:

Yesterday, Tom was going for a walk. He used the time to think about his future.

In this little example, both "Tom" and "He" refer to the same entity. Relating "Tom" directly with the last sentence makes the relation clearer, especially for computers. This way, coreferences greatly support the structuring and clustering of information and therefore help to retrieve similar information.

Based on the last lines, the tags of table 3.7 for annotating text have been identified.

Tag	Description
Requirement Document	This tag marks the beginning and ending of a requirement document.
Requirement	This tag marks the beginning of a requirement. It has an ID attribute for its unique identification. Additionally it could refer to a parent requirement, i.e. a requirement, which is above it in the requirement hierarchy.
Non-functional requirement	This tag marks the beginning of a non-functional requirement. In addition to the attributes of the normal requirement, a NF-requirement contains a type-attribute, which specifies the type of the following information, e.g. "OperatingSystem", "RAM" etc.
Coreference	The probably most important tag. It marks words, which relate to other words. Besides an ID attribute, it contains a source-attribute, which refers one coreference tag to other coreference tags. Its value is the ID of the other coreference tag.

Table 3.7: Tags for annotation of requirements

3.2.2 Semi-automatic annotation using NLP

This thesis already referred to the point of semi-automatic annotation. For this, the computer must be able to extract specific information from the text. Natural language is a very complex domain which is difficult for machines to understand (in a semantic manner). Due to this, natural language processing (NLP) is a huge field of science which tries to take natural language text and find certain information in it. Basically, the following steps are necessary in an NLP chain:

- Sentence detection: This describes a technique to split a text into its single sentences.

- Token extraction: Based on the single sentences, these are split into their tokens.
- Treebank Parsing: A Treebankparser tries to assign every token its specific meaning, i.e. the NLP tool decides, if the token is a verb, an adverb, an adjective etc.
- Namefinding: This does exactly what the name proposes - names of persons, corporations etc. are being retrieved.
- Coreference resolution: The last step is the detection of potential coreferences.

Due to the complexity of the domain, many of these tasks have only a very low correct detection ratio and / or only encompass certain parts of their particular field. The first two steps are known to work pretty well, whereas the treebank parsing is a step, which tries with statistical methods to assign the specific types of the words to the single tokens (see [23]). Depending on the probability, this step sometimes works out quite well, but sometimes does not. As the NLP processing method can be seen as some kind of dataflow where the output of one action is the input of the next one, the accuracy of the results is reduced by the number of errors in earlier phases. Depending on the type of the implementation, namefinding could work quite well. The last point (coreference resolution) is the most difficult task and most often only works on so-called named-entity relations, i.e. only coreferences between already found names and pronouns can be detected. Section 3.2.2 gives a short introduction into general coreference detection, the relevant technical terms and its problems.

Results which can now be used in the semi-automatic detection are name-finder and of course the coreference results. The idea behind using the results of the name-finder is that many simple NF-requirements have name-like characteristics, i.e. these are short strings / combination of words like "1024MB RAM" or "Windows XP". If those NF requirements have been found, they can simply be annotated with the corresponding tags. Still, the NLP process is not yet in a stage, where it could completely automatize the whole process. Therefore, the input of the user is still required, either to correct wrongly annotated information or to annotate information, which has not been identified at all.

Coreference detection

A coreference has been defined by Mitkov ([25]) as two textual entities which refer to the same real world object. More formally, this can be expressed by saying α_1 and α_2 corefer, it $referent(\alpha_1) = referent(\alpha_2)$. Parsers for coreference detection (sometimes also referred to as coreference resolution or dependency parsing) can be split into two different categories, namely projective only and non-projective parsers. The latter also incorporate the first one, whereas projective parsers are easier to develop. Projective and non-projective refers to the kind of relation that coreferences can have, i.e. given a

dependency tree (i.e. a directed graph which holds the dependencies between different words of a text), the nodes of a subtree must form a continuous interval, where an interval (with endpoints i and j) is the set

$$[i, j] = \{k \in V | i \leq k \text{ and } k \leq j\}$$

where V is the set of nodes in the dependency tree (more details can be seen in, e.g., [18]).

Figure 3.2 shows a projective dependency tree, whereas figure 3.3 shows a non-projective subtree. The crossing edges in this case mark a non-continuous interval and therefore a non-projective dependency tree.

Many dependency-based parsers are restricted to projective dependency structures, which greatly facilitate the parsing process ([18]). However, it is known that many syntactic constructions can only be represented by non-projective structures. This is particularly true for languages like Czech, which exhibit a free or very flexible word order.

The "Message Understanding Conference" (MUC) gave another definition of a coref-

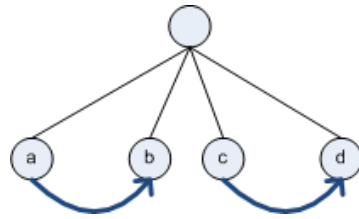


Figure 3.2: Projective coreferences

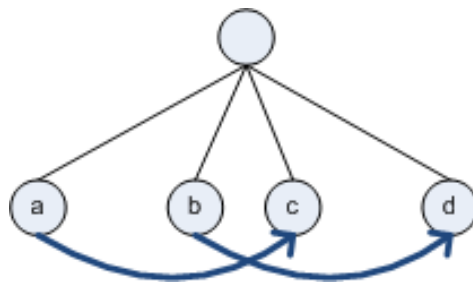


Figure 3.3: Non-projective coreferences

erence relation ([10]):

An identity-of-reference relation between two textual elements known as Markables.

The word "Markables" is the source of the next problem. A markable can be e.g. a noun phrase or a proper name. A well-known approach is the so-called pipelined approach which takes a text and starts by splitting it into single sentences. These are then being tokenized and tagged according to their syntactic positions in the text corpus. Next, the noun phrase and named entity recognition as well as nested noun phrase extraction takes place, before, finally, a semantic class determination can be induced.

Still, the noun phrase recognition process is a highly complex task, which is known to be error prone. Many of today's approaches rely on existing training data to capture correct noun phrases. However, there are often several uncertainties which can have an impact on the final result.

3.2.3 Basic ontology structure

The idea behind this work is that with the help of technologies evolved from the semantic web field, the matching of artefacts is supported. Therefore, after the identification of all relevant information, an ontology has been developed, which is capable of keeping this information in a structured and reasonable way. Through the use of an ontology, inference machines can be used to realize the similarity matching process. The basic ontology itself is presented in figure 3.4. As it can be seen, the ontology consists of two parts: a basic artefact description and a non-functional requirement description, which will now be explained in more detail.

Basic artefact description

The basic artefact description consists of three basic elements: The 'artefact' itself, the 'Requirement' and 'Word'. An artefact is the simple representation of everything which could be an artefact, either a method, a class etc. An artefact itself is described by ('describedBy') its requirements ('Requirement'), which can have an arbitrary number of sub-requirements ('hasSubRequirement'). Every 'Requirement' in turn is defined by ('definedBy') words ('Word'). A word itself can be related to ('relatedTo') or equal ('equals') another word.

Non-functional requirement description

Non-functional requirements can have more detailed types. There are two reasons for this:

- Performance - The ontology matching algorithm described later on in this work, is a very time consuming process. Therefore, a more precisely defined NF-requirement type should accelerate the matching of NF-requirements and therefore the similarity-matching as a whole, because only NF-requirements of the

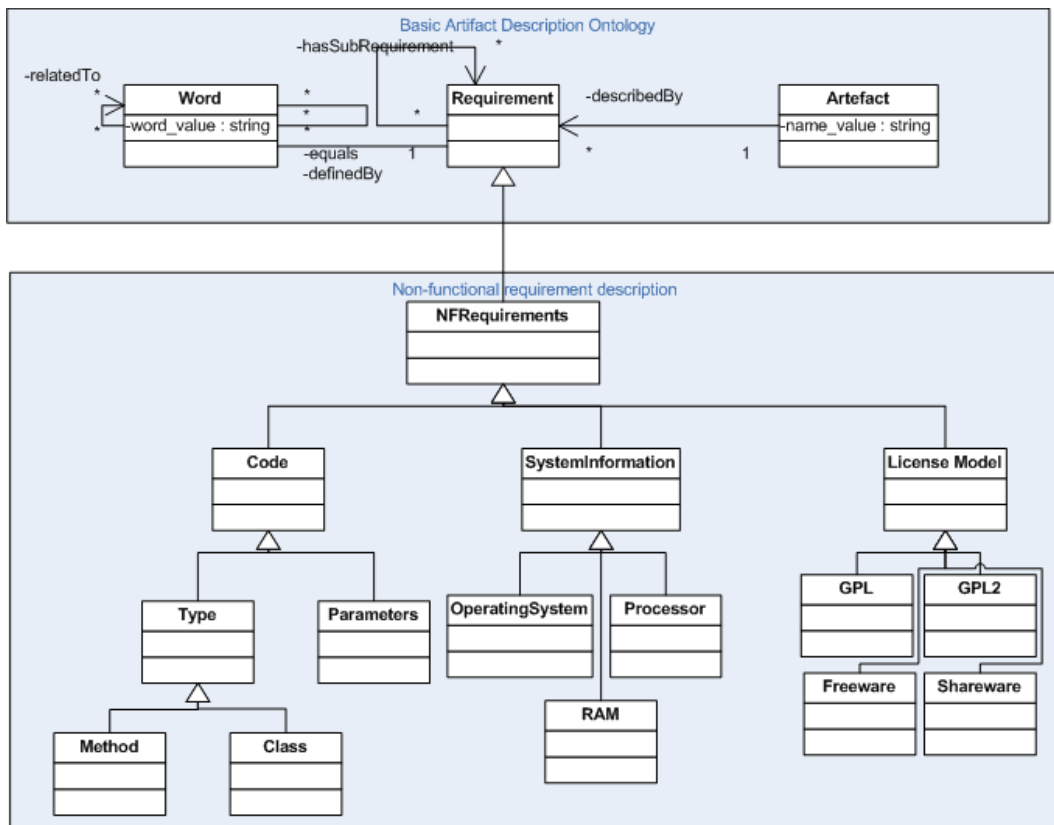


Figure 3.4: TBox structure of the basic ontology

same type like 'OperatingSystem' or 'RAM' will be considered for further comparison.

- Better comparison - NF-requirements can often consist of simple values like names, e.g. 'Core 2 Duo with 2.0 GHz' and 'C2D with 2400 MHz'. The comparison of both values is a difficult and often inaccurate task for computers to do, especially, if they do not know what kind of data they have to deal with. A big help could therefore be to explicitly state that the example mentioned is of the type 'Processor'. Now, an additional comparison method could be called, which certifies that both strings are highly similar.

The ontology, as it can be seen in figure 3.4, only shows a small example (due to the prototypical character of this work) of what kind of types could be modelled into the ontology. Further points could be, for example, a 'Runtime' object with its corresponding extension method.

3.2.4 Similarity ontology structure

For the purpose of giving a better overview, the last section only covered the general basic structure of the ontology. This section will now show and or describe all possible values, which are of importance to the similarity matching process. These additional relations are:

- `isSimilarWord`: This relation indicates if two words seem to be similar (how this similarity is determined will be shown later on in 3.3.3)
- `matchingNFRequirement`: If two NF-requirements match, this relation is inserted into the ontology.
- `haveMatchingNFRequirements`: If two artefacts have a certain amount of matching NF requirements, this relation is inserted between both artefacts.
- `isStructuralMatching`: This is inserted between two words, if a structural match has been found (see 3.3.4 for more details about what a structural match is).
- `isSimilarRequirement`: If two requirements are known to be similar, both will be connected via this edge.
- `isSimilarArtefact`: This is computed the same way as `isSimilarRequirement` and shows, if two artefacts are known to be similar.

Additionally, two new classes will be inserted if the similarity or satisfaction-value calculation for elements of either type "Word", "Requirement" or "Artefact" is done:

- **Similarity:** The similarity concept contains the value of the similarity calculation, i.e. based on the calculation of the similarity between for example two elements A and B, a new individual S of type "Similarity" will be inserted into the ontology. A relation of type "hasSimilarity" will be inserted between A and S. S refers to B via a relation of type "similarityRefersTo". Note, that every word / requirement / artefact tuple only contains one similarity value. Also, similarity is a symmetric value, i.e. A has the same similarity to B as B has to A.
- **Satisfaction:** The satisfaction concept contains the value of the satisfaction calculation, i.e. based on the calculation of the satisfaction between two elements A and B, a new individual S of type "Satisfaction" will be inserted into the ontology. A relation of type "hasSatisfaction" will be inserted between A and S. S refers to B via a relation of type "satisfactionRefersTo". Again, every element tuple contains only one satisfaction value, but compared to the similarity value, satisfaction is not symmetric.

The calculation of both values will be explained later in section 3.3.

3.2.5 Text to ontology transfer

One point, which has been out of focus so far is how the text is correctly inserted into the ontology. Normally, text forms some kind of chain. Nearly every word is therefore followed by another one. If you look at the last sentence, "Nearly" is followed by "every", which is followed by "word" and so on. One problem is, how to split the text into its single tokens, because the similarity matching process will only work on the single tokens themselves. The solution to this problem is in this case pretty straightforward: NLP tools support the tokenization of whole texts and do a better job of recognizing for example abbreviations (i.e. abbreviations like "M. Schneider" could also be detected as the end of a sentence) than a simple reg-ex approach would do, i.e. simply splitting the text by its white spaces. Still, not every word is of importance to the similarity matching process. Words like "a", "an", "as" etc. are considered to be so-called stop-words, i.e. words which do not contain any relevant information ([22]). Before any insertion process into the ontology takes place, these stop-words will be eliminated from the textual corpus. After that the list of tokens can be simply inserted into the ontology. Of course, the creation of information which is indicated by the tags has to be considered. Therefore, a stack-like approach should be used: All tags are inserted into the list of tokens at their corresponding position. Then the insertion algorithm starts at the very beginning of this list and works its way through the list. Every time a tag is being found in the list, this tag will be put on one of two stacks:

- **Requirement stack:** This stack only holds those tags which are of type requirement.

- Coreference stack: This stack only holds those tags which indicate any kind of coreference.

Depending on which tag element is on top of this stack, additional information will be added to the newly inserted word. For example, depending on the requirement which lies on top of the requirement-stack, a word will be added to exactly this requirement, i.e. a new "definedBy" relation is inserted between the requirement and the new word. If there is also an element on top of the coreference stack, additional information about this coreference will be inserted into the ontology as well. This is done by inserting an "equal" relation between the corresponding two words (see 3.2.3). It is important to note, that every individual, which is being inserted into the ontology, must be uniquely identifiable (see 3.3.1 for more details).

3.3 Artefact comparison algorithm

This section specifies the key idea behind this thesis. As already mentioned in the previous sections, the idea was to semi-automatically annotate text, transfer it into an ontology and then look for similarities with other ontologies, which use the same TBox. The similarity-matching process can be seen as a bottom-up approach, which starts with the comparison of the smallest possible element (a word) and works its way up the top-node (an artefact). Therefore, first the basic preparations for making the algorithm work will be shown. After that, every single one of the four theoretical phases will be explained in detail, showing what they do, what the corresponding rules look like and what the necessary inbuilt are.

3.3.1 Preparation

Before starting with the phases themselves, the first basic step is to unite both ontologies, i.e. join both artefact description ontologies in one ontology. This is necessary for the reasoning to work properly. It is very important that every item in both ontologies is uniquely identifiable, i.e. has a unique namespace identification, even after the union of both ontologies. If this was not the case, the reasoning results could be unpredictable and simply wrong, because two elements, which initially belong to different artefacts but share the same namespace, would be treated the same way or probably not at all. If this precondition is met, both ontologies can be joined and the inference mechanism will begin.

3.3.2 Phase 1 - Non-functional Requirement matching

The first phase of the whole similarity matching process is also the only one which could stop the ongoing similarity calculation. In this phase, all NF requirements of

one artefact are compared to the NF-requirements of the other artefact. This is done by comparing the words belonging to an NF-requirement on a string basis. The idea behind this is that NF-requirements often only consist of shorter string descriptions like "Windows XP". This assumption can greatly reduce the amount of comparison calculations needed and therefore restrict the number of artefacts which need to be compared.

The rules necessary for the reasoning mechanism of this phase are shown in table 3.8 and 3.9. The first rules are the similar word ones. Those take every word which is associated with an NF-requirement and compare it to the word of an NF-requirement of the other artefact. This comparison is done using the algorithm of 3.4.3. If this value is above a certain threshold both words seem to be similar and therefore a new relation "isSimilarWord" will be inserted between both words. This step is done for all words of every NF-requirement. After having finished the word-similarity calculation, the similarity of every NF-requirement tuple is calculated (rule "Similar NF requirement"). As can be seen in the rule, there is a method *AreSimilarNFRequirements(R1, R2, isSimilarWord)*. As the basic algorithm, needed for the computation of similarity in this method, will be used several times throughout the whole matching process, it will be explained in more detail in 3.4.1.

If the NF requirements of both artefacts have a high similarity value (calculated via *AreSimilarNFArtefacts(A, B, haveMatchingNFRequirements)*) the process will continue, else it will stop the matching calculation between both artefacts and continue with the comparison of two other artefact descriptions (rule "Have matching NF requirements 1"). If one or probably none of both artefacts contains NF-requirements, the similarity matching algorithm will continue its work with phase 2, because nothing can be said about the matching of both artefacts regarding their NF-requirements (rule "Have matching NF requirements 2 / 3").

3.3.3 Phase 2 - Syntactic / Semantic matching

After phase 1 is finished and the NF-requirements seem to match to a certain degree, phase 2 can begin. This phase now begins with a lot of preparation rules (transitivity, inverse transitivity etc.). Next, it starts working on the normal text and is therefore a very time-consuming process, because every word of every non NF-requirement of one artefact is compared to every word of the other artefact. For comparison, two different ways will be used additionally. The first one is based on a simple string to string comparison (see 3.4.3), whereas the second one tries (see 3.4.2) to find words with a similar meaning or, more generally, a high semantic relatedness. Both similarity metrics are calculated for one word. If both metrics yield a specific, user-defined threshold, the metric with the highest similarity value will be taken and its value inserted into the ontology. In case of only one value being equal or bigger than its corresponding

<i>Part of description</i>	$A : \text{describedBy } B \wedge B : \text{definedBy } C \rightarrow C : \text{partOfDescriptionOf } A$
	If an artefact A is described by a requirement B and this requirement is defined by a word C, then C is part of the description of the artefact A.
<i>Inverse defined by</i>	$A : \text{definedBy } B \rightarrow B : \text{inv_definedBy } A$
	If a requirement A is defined by a word B, the B has an inverse definedBy relation to A.
<i>Similar Word Symmetry</i>	$A \text{ isSimilarWord } B \rightarrow B \text{ isSimilarWord } A$
	If the word A is similar to B, then B is also similar to A
<i>Similar Word</i>	$A \text{ partOfDescriptionOf } X \wedge A \text{ Word_Value } VA \wedge A \text{ inv_definedBy } N \wedge N \text{ type } NFRequirement \wedge B \text{ partOfDescription } Y \wedge B \text{ inv_definedBy } M \wedge B \text{ Word_Value } VB \wedge M \text{ type } NFRequirement \wedge X \neq Y \wedge \text{IsSimilarString}(VA, VB) \rightarrow A \text{ isSimilarWord } B$
	If the word A is part of the description of artefact X and A has a concrete value VA and A belongs to the NF-requirement N and there is also a word B, which belongs to an artefact Y and B has a concrete value VB and B belongs to the NF-requirement M and the artefacts X and Y are not the same and the concrete word values VA and VB are similar, when the words A and B are similar.
<i>Similar NF requirement symmetry</i>	$R1 \text{ isSimilarNFRequirement } R2 \rightarrow R2 \text{ isSimilarNFRequirement } R1$
	If the NF-requirement R1 is similar to R2, then R2 is also similar to R1.
<i>Similar NF requirement</i>	$A \text{ describedBy } R1 \wedge B \text{ describedBy } R2 \wedge R1 \text{ type } NFRequirement \wedge R2 \text{ type } NFRequirement \wedge A \neq B \wedge \text{AreSimilarNFRequirements}(R1, R2, \text{isSimilarWord}) \rightarrow R1 \text{ isSimilarNFRequirement } R2$
	If the artefact A is described by the NF-requirement R1 and the artefact B is described by the NF-requirement R2 and A and B are not the same artefacts and R1 and R2 are similar NF-requirements, then insert the isSimilarNFRequirement relation between R1 and R2.

Table 3.8: Rules for phase 1 part 1

<i>Have matching NF requirements symmetry</i>	$A \text{ haveMatchingNFRequirements } B \rightarrow B \text{ haveMatchingNFRequirements } A$
	If A and B have matching NF-requirements, then B and A also have matching NF-requirements.
<i>Have matching NF requirements 1</i>	$A \text{ type artefact} \wedge B \text{ type artefact} \wedge A \neq B \wedge \text{AreSimilarNFArtefacts}(A, B, \text{haveMatchingNFRequirements}) \rightarrow A \text{ haveMatchingNFRequirements } B$
	If A and B are different artefacts and the both have similar NF-requirements, A and B have matching NF-requirements.
<i>Have matching NF requirements 2</i>	$A \text{ type artefact} \wedge B \text{ type artefact} \wedge A \neq B \wedge B \text{ describedBy } Y \wedge \neg \exists (A \text{ describedBy } \text{NFRequirement}) \rightarrow A \text{ haveMatchingNFRequirements } B$
	If A and B are different artefacts and A has no NF-requirement description, but B has at least one NF-requirement, still both A and B have matching NF requirements.
<i>Have matching NF requirements 3</i>	$A \text{ type artefact} \wedge B \text{ type artefact} \wedge A \neq B \wedge \neg \exists (B \text{ describedBy } \text{NFRequirement}) \wedge \neg \exists (A \text{ describedBy } \text{NFRequirement}) \rightarrow A \text{ haveMatchingNFRequirements } B$
	If A and B are different artefacts and neither A nor B have a NF-requirement description, still both A and B have matching NF requirements.

Table 3.9: Rules for phase 1 part 2

threshold, this one is taken. This kind of comparison requires both metrics to scale on the same range, in this case on the interval $[0, 1]$.

3.3.4 Phase 3 - Structural matching

The similarity matching process would already find similar texts, if it consisted only of phase 2. However, the problem would be that both texts still could not contain information about the same artefact. The cause for this potential problem would be that synonyms and similar words can be found throughout the whole text. The probability for finding similar words in a text additionally increases with the size of the requirement document, because the more words a text corpus contains, the more occurrences of different words will exist. Therefore it is important to implement a structural matching, which will specify that there are not only similar words, but that these also exist in a similar content.

Given two words w_1 and w_2 of artefact A_1 and two other words y_1 and y_2 of artefact A_2 , a structural match between w_1 , w_2 , y_1 and y_2 will be found, if the following conditions are met:

1. w_1 and w_2 must be connected by a *trans_relatedTo* or a *trans_inv_relatedTo* relation
2. y_1 and y_2 must be connected by a *trans_relatedTo* or a *trans_inv_relatedTo* relation
3. The distance d_w between w_1 and w_2 must be below a certain threshold.
4. The distance d_y between y_1 and y_2 must be below a certain threshold.
5. w_1 must have an *isSimilarWord* relation to either y_1 or y_2 .
6. w_2 must have an *isSimilarWord* relation to either y_1 or y_2 .
7. All four words must be connected to at least one corresponding counterpart in the other ontology, e.g. $w_1 \rightarrow y_1$ and $w_2 \rightarrow y_2$.
8. The ratio $\frac{\min(d_w, d_y)}{\max(d_w, d_y)}$ must be above a certain threshold. This indicates that both words in one ontology have a similar relatedness as their counterparts in the other ontology. In case of $w_1 \rightarrow y_2$ and $w_2 \rightarrow y_1$, where w_1 is related to w_2 and y_1 is related to y_2 (a cross reference), the user defined threshold for the ratio changes.

The distance between two words w_1 and w_2 is the number of steps, which is required to get from the first to the last word, using the *relatedTo* or *inv_relatedTo* relation. This metric is calculated by using the algorithm from 3.4.4. If all these conditions are met, a new relation of type *isStructuralMatch* will be inserted between both of the similar word pairs. The corresponding rules are shown in table 3.11.

<i>Inverse related to</i>	$A : relatedTo B \rightarrow B : inv_relatedTo A$ If A is related to B, then B is inversely related to A
<i>Inverse described by</i>	$A : describedBy B \rightarrow B : inv_describedBy A$ If a requirement A is described by a word B, the B has an inverse describedBy relation to A.
<i>Transitive related to 1</i>	$A : relatedTo B \rightarrow A : trans_relatedTo B$ If a word A is related to another word B, then A is also transitively related to B.
<i>Transitive related to 2</i>	$A : trans_relatedTo B \wedge B : trans_relatedTo C \rightarrow A : trans_relatedTo C$ If a word A is transitively related to another word B and B is also transitively related to a word C, then A is also transitively related to C.
<i>Transitive inverse related to 1</i>	$A : relatedTo B \rightarrow B : trans_inv_relatedTo A$ If a word A is related to another word B, then B is also transitively and inversely related to A.
<i>Transitive inverse related to 2</i>	$A : trans_inv_relatedTo B \wedge B : trans_inv_relatedTo C \rightarrow A : trans_inv_relatedTo C$ If a word A is transitively and inversely related to another word B and B is also transitively and inversely related to a word C, then A is also transitively inversely related to C.
<i>Coreference 1</i>	$A : equals B \wedge B : relatedTo C \rightarrow A : relatedTo C$ If a word A equals B and B is related to C, then A is also related to the word C. For a clearer understanding imagine the slightly changed example from the previous chapter: "Yesterday, Tom was going for a walk. During the time, he thought about his future.", where "Tom" and "he" refer to the same entity. This rule adds the information, that "Tom" is also related to "used" in the second sentence.
<i>Coreference 2</i>	$A : equals B \wedge B : inv_relatedTo C \rightarrow C : relatedTo A$ If a word A equals B and B is inversely related to C, then C is also related to the word A. To demonstrate this rule, the example from the first coreference rule is taken: "Yesterday, Tom was going for a walk. During the time, he thought about his future.", where "Tom" and "he" refer to the same entity. This rule adds the information, that "time" is also related to "Tom".
<i>Similar Word</i>	$A \ partOfDescriptionOf \ N \wedge B \ partOfDescriptionOf \ M \wedge A \ inv_definedBy \ R1 \wedge B \ inv_definedBy \ R2 \wedge A \ WordValue \ VA \wedge B \ WordValue \ VB \wedge \neg(R1 \ type \ NFRequirement) \wedge \neg(R2 \ type \ NFRequirement) \wedge N \neq M \wedge isSimilar(va, vb) \rightarrow A \ isSimilarWord \ B$ If the word A is part of the description of the artefact N and B is part of the description of artefact M and A and B do not belong to a NF-requirement and A and B are similar, then A is a similar word to B.

Table 3.10: Rules for phase 2

<i>Structural Match Symmetry</i>	$A \text{ isStructuralMatch } B \rightarrow B \text{ isStructuralMatch } A$
	If the word A has a structural match with B, then B also has a structural match with A
<i>Structural Match</i>	$A \text{ isSimilarWord } M \wedge B \text{ isSimilarWord } N \wedge A \text{ partOfDescriptionOf } Art1 \wedge B \text{ partOfDescriptionOf } Art1 \wedge M \text{ partOfDescriptionOf } Art2 \wedge N \text{ partOfDescriptionOf } Art2 \wedge Art1 \neq Art2 \wedge A \text{ trans_relatedTo } B \wedge M \text{ trans_relatedTo } N \wedge dist(A, B) \leq d \wedge dist(M, N) \leq d \wedge ratio(dist(A, B), dist(M, N)) \geq k \rightarrow A \text{ isStructuralMatch } M \wedge B \text{ isStructuralMatch } N$
	If the word A has a similarity with M and B has a similarity with N and A and B belong to artefact Art1 and M and N belong to artefact Art2 and Art1 is not Art2 and A is transitively related to B and M is transitively related to N and the distance between A and B is smaller than or equals d and the distance between M and N is smaller than or equals d and the ratio of both distances is greater than or equals k, then A has a structural match with M and B has a structural match with N.
<i>Structural Cross Match</i>	$A \text{ isSimilarWord } M \wedge B \text{ isSimilarWord } N \wedge A \text{ partOfDescriptionOf } Art1 \wedge B \text{ partOfDescriptionOf } Art1 \wedge M \text{ partOfDescriptionOf } Art2 \wedge N \text{ partOfDescriptionOf } Art2 \wedge Art1 \neq Art2 \wedge A \text{ trans_relatedTo } B \wedge M \text{ trans_inv_relatedTo } N \wedge dist(A, B) \leq d \wedge dist(M, N) \leq d \wedge ratio_{cross}(dist(A, B), dist(M, N)) \geq k_{cross} \rightarrow A \text{ isStructuralMatch } M \wedge B \text{ isStructuralMatch } N$
	If the word A has a similarity with M and B has a similarity with N and A and B belong to artefact Art1 and M and N belong to artefact Art2 and Art1 is not Art2 and A is transitively related to B and M is transitively related to N and the distance between A and B is smaller than or equals d and the distance between M and N is smaller than or equals d and the cross ratio of both distances is greater than or equals k_{cross} , then A has a structural match with M and B has a structural match with N.

Table 3.11: Rules for phase 3

3.3.5 Phase 4 - Requirement / artefact similarity computation

The final phase now takes the results of the previous stages and computes both the satisfaction and the similarity value. This is done in both cases by using the algorithm from 3.4.1. Compared to phase one, this time, different relation types are taken to calculate both values. Table 3.12 shows the rules that have been used.

With the end of phase 4, a single value for both similarity and satisfaction of two

<i>Similar Requirement Symmetry</i>	$R1 \text{ isSimilarRequirement } R2 \rightarrow R2 \text{ isSimilarRequirement } R1$
	If the requirement R1 is similar to requirement R2, then R2 is also similar to R1
<i>Similar Requirement</i>	$A \text{ describedByt } R1 \wedge B \text{ describedBy } R2 \wedge A \neq B \wedge \text{AreSimilarRequirements}(R1, R2, \text{isStructuralMatch}) \rightarrow R1 \text{ isSimilarRequirement } R2$
	If the artefact A is described by requirement R1 and B is described by requirement R2 and both requirements are similar, regarding the "isStructuralMatch" relation, R1 is similar to R2
<i>Similar artefact Symmetry</i>	$A1 \text{ isSimilarartefact } A2 \rightarrow A2 \text{ isSimilarartefact } A1$
	If the artefact A1 is similar to artefact A2, then A2 is also similar to A1
<i>Similar artefact</i>	$A1 \text{ type artefact } \wedge A2 \text{ type artefact } \wedge A1 \neq A2 \wedge \text{AreSimilarartefacts}(A1, A2, \text{isSimilarRequirement}) \rightarrow A1 \text{ isSimilarartefact } A2$
	If there are two artefacts A1, A2 and A1 is not A2 and A1 is similar to A2, regarding the "isSimilarRequirement" relation, then A1 is similar to A2

Table 3.12: Rules for phase 4

artefacts has been computed. The higher these values are, the higher is either the similarity or the satisfaction of one artefact towards another. Based on this computer generated information, the user should be able to make a quicker and better decision about which component to use.

3.4 Additional algorithms

As already mentioned before, different algorithms from other fields have been used for one, Lin ([21]), which makes use of Wordnet, and Jaro-Winkler ([37]) for the string based similarity measurement.

3.4.1 Graph based similarity measure

This section describes in detail, how the algorithm used to compute the similarity between the same level of different artefacts, works. A level of the ontology of section 3.2.3 is either the "Requirement" or the "Artefact" level. Both have in common that they are defined by a number of "children", i.e. in case of an artefact its requirements or in case of a requirement its words. If the hierarchy is seen as a tree-like structure, this algorithm will not work on the leaves of the tree, but only on the inner nodes. A formal definition of the algorithm follows:

The tree consists of a set N of nodes. L defines the number of leaves. C is the set of children of a node $n \in N \wedge n \notin L$. The similarity is defined as

$$sim(n_1, n_2, r) = \frac{\sum_{c_i \in C_1, c_j \in C_2} totalrelations(c_i, c_j, r)}{numitems(C_1) + numitems(C_2)}$$

where $n_1 \in N \wedge n_1 \notin L$, $n_2 \in N \wedge n_2 \notin L$, r defines a certain type of relation, which exists between the items in C , C_1 is the set of children of n_1 , C_2 is the set of children of n_2 , $totalrelations(c_i, c_j, r)$ counts the occurrences of r between two children c_i and c_j , $numitems(C_1)$ returns the total number of items in C_1 . Notice, that $sim(n_1, n_2, r) = sim(n_2, n_1, r)$.

An additional factor, which is being calculated, is the so called "satisfaction". Satisfaction describes a ratio of elements, which are satisfied by a relation to an element of another set. The satisfaction is defined as

$$sat(n_1, n_2, r) = \frac{\sum_{c_i \in C_1, c_j \in C_2} numrelation(c_i, c_j, r)}{numitems(C_1) + numitems(C_2)}$$

where $numrelation(c_i, c_j, r)$ only counts such relations r , which are going from c_i to c_j (in contrast to $totalrelations$, which counts all r between c_i and c_j). This is also the cause for satisfaction not being symmetric, i.e. $sat(n_1, n_2, r) \neq sat(n_2, n_1, r)$

3.4.2 Similarity measure based on Wordnet

Measuring the semantic relatedness between different words or concepts is an easy task for humans. Computers have more difficulties in replicating this process and determining the similarity between two words. Over the last decades different approaches and algorithms have been developed to measure the semantic relatedness between concepts. Words can have different relations to each other. There are for example IS-A (like hypernym-hyponym or part-whole), associative or equivalence relations (see [15]). The most common type which is also seen as the most important relation type is the hierarchical one, because it can be mapped well onto the human way of thinking.

Whereas knowledge-free approaches exist, which simply rely on the corpus data, state-of-the-art algorithms use machine-readable dictionaries like WordNet ([8]) to calculate

the semantic similarity.

The approach which will be used in this work has been developed by Lin ([21]) and will be described in more detail on the following page.

Semantic similarity

Lin first gives a general introduction of his sense of similarity, which can be applied to many different domains (not only strings):

$$sim(A, B) = \frac{\log P(common(A, B))}{\log P(description(A, B))}$$

where $common(A, B)$ is a proposition that states the commonalities of two terms A and B and $description(A, B)$ defines, what A and B are. $P(\dots)$ is a function, which gives the probability of a certain statement.

The semantic similarity itself refers to the semantic similarity of two classes C and C' in a taxonomy, i.e. given an $x \in C$ and $x' \in C'$, the amount of information, contained in $x \in C$ and $x' \in C'$ is

$$-\log P(C) - \log P(C')$$

where $P(C)$ and $P(C')$ are the probabilities of a randomly selected object belonging to C and C' . Given the tree of the taxonomy and the conditions, that $x_1 \in C_1$ and $x_2 \in C_2$, the commonality between x_1 and x_2 is $x_1 \in C_0$ and $x_2 \in C_0$, where C_0 is the most specific class, which subsumes C_1 and C_2 . Following this, the formula is

$$sim(x_1, x_2) = \frac{2 * \log P(C_0)}{\log P(C_1) + \log P(C_2)}$$

The similarity between this formula and the general formula for similarity can easily be seen: The probability $\log P(C_0)$ specifies the information, which both concept nodes C_1 and C_2 have in common, whereas the sum of $\log P(C_1)$ and $\log P(C_2)$ defines the description of all information of C_1 and C_2 .

Figure 3.5 shows a little fragment of the inner WordNet Taxonomy. The numbers, attached to each node, represent $P(C)$. Based on the formula above, the similarity between "Hill" and "Coast" is:

$$sim(Hill, Coast) = \frac{2 * \log P(geological-formation)}{\log P(Hill) + \log P(Coast)} = 0.59$$

Compared in a benchmark, developed by Miller & Charles, with other WordNet-based similarity measures, Lin's approach showed an overall correct similarity measurement of 0.834 (compared to a humans judgement, which is 1.0).

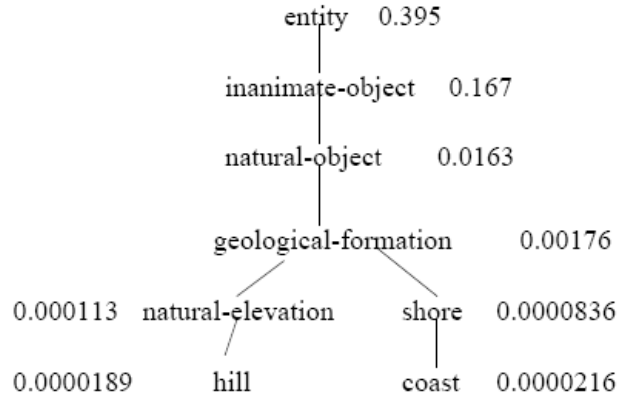


Figure 3.5: A WordNet fragment

3.4.3 String based similarity measure

A problem when dealing with different but probably equal texts is that one and the same word may be misspelled in one or both of the texts. The WordNet based similarity approach can only identify semantic closeness between correctly spelled words, but not between misspelled ones. Additionally, it could happen (though unlikely), that one and the same word is misspelled in both texts. Still, it should be possible to find such words. Therefore, a second algorithm will be used which is based on a simple string comparison. One first approach in this work relied on an algorithm by A. Monge and C.Elkan ([26]), but it delivered unsatisfying results for certain tuples like 'Athlon' and 'It' - their similarity value was 0.8 on a scale from 0 to 1. The new approach is based on the Jaro-Winkler metric ([37]), which delivers more promising results. Additionally, it seems to be intended especially for shorter strings as is the case with the data in the ontology ([3]). The algorithm is described in more detail in the following section.

Jaro-Winkler string similarity algorithm

The algorithm of Winkler is based on the basic algorithm of Jaro:

$$\Phi_j(str_1, str_2) = \frac{1}{3} \left(\frac{com(str_1, str_2)}{len(str_1)} + \frac{com(str_1, str_2)}{len(str_2)} + 0.5 * \frac{transpos(str_1, str_2)}{com(str_1, str_2)} \right)$$

where $com(str_1, str_2)$ computes the number of common characters between two strings, $len(str_1)$ returns the length of a string and $transpos(str_1, str_2)$ returns the number of characters, which would have to be changed to turn one string into the other. $com(str_1, str_2)$ considers two characters to be in common, if

$$\frac{max(len(str_1), len(str_2))}{2} - 1$$

Winkler took the basic function Φ_j and slightly changed it ([36]):

$$\Phi_n(str_1, str_2) = \Phi_j(str_1, str_2) + i * 0.1 * (1 - \Phi_j(str_1, str_2))$$

where i is the number of equal characters in the beginning of both strings with a total maximum of four matches. This way, strings, which have equal characters in the beginning, automatically have a higher similarity due to the higher probability of a possible match.

3.4.4 Distance measurement in an ontology

Several of the rules need the distance between two words which have been inserted into the ontology. There are many known algorithms for the shortest path problem, one of the most famous is Dijkstra. Although AStar is known to have better average runtime values, it cannot be used in this case, because no information is available about the distance of one arbitrary node in the ontology to the target node of the path, i.e. there is not yet a possibility, to compute any additional heuristics. Therefore, Dijkstra will be presented in the following section just for completeness (the description is taken out of [13]).

Dijkstra

An ontology with its relations between the different nodes can be seen as a graph G with a set of vertices V and a set of edges E , where every vertex is comparable to a node in the ontology and every edge $\in E$ can be seen as a relation of one certain objectproperty type between two nodes in the ontology (i.e. the path in the ontology is calculated along one objectproperty type (e.g. "relatedTo") and is not using two or more objectproperty types at once).

The Dijkstra algorithm works by calculating the shortest distance $d(s, c)$ from the initial vertex s to the current vertex c , where $s \in V$ and $c \in V$. The initial distance value of s is 0. To keep track of all vertices, the algorithm works with two lists of vertices, C and F . C holds all those vertices, which are known to already have the shortest path to s and is therefore initially empty. F instead holds all other vertices. In each step, the vertex c with the lowest distance $d(s, c)$ in F is taken and removed from F . After that, its edges are being *relaxed*, i.e. for every direct (i.e. reachable via one outgoing edge of c) neighbour n of c the distance is newly calculated. If this new value is smaller than the old distance value, this means, that the path could be optimized and n will be given the new distance value. In this case, the algorithm will terminate if the vertex c , being moved from F to C equals the target node t , i.e. the vertex, which is the end of the path to be computed.

Chapter 4

Realization

Chapter 3 (3.3) described the basic idea and the concept, which lies behind this thesis. The results have been taken and used for a prototypical implementation, whose architecture will be described in the next few sections. As the aim is to develop just a prototypical demonstration of the concept, the simple waterfall methodology ([12]) has been used. This chapter is therefore structured the same way, laying its focus on three specific areas: Requirements, Design and Realization. Section 4.1 first gives an insight into the requirements this prototype had to fulfill, before the evaluation of necessary components will be shown (4.2). Next, the static design will be explained (4.3). As a software is a highly dynamic system, the dynamic aspects will be presented in (4.4). The last section finally explains the realization of the prototype (4.5).

4.1 Requirements

Section 3.1.2 has already shown what a SemRE tool could look like to fully support the user. As this work just shows the prototypical implementation of the algorithm, many scenarios have been left out. Basically, the prototype just shows the basic feasibility of the semi-automatic annotation of text as well as the similarity matching between different components, using the algorithm from section 3.3. The requirements this prototype has to fulfill are:

Loading of text files

The prototype must be capable of loading simple text files. The text in these files can either be fully annotated with the corresponding tags, or it is a plain text file.

Automatic annotation of text

Text files which do not contain a proper XML schema are being defined as not annotated. In this case, the SemRE prototype should start a complete NLP process which automatically identifies as much information as possible.

Saving an annotated text file

The user must be able to save his newly annotated text file. Although the annotation should be done in a pretty fast way, it is still a time consuming task which can lead to errors. Therefore the user should not have to annotate a text more than once.

Annotation of text - visually

As the NLP process is error prone and sometimes simply detects the wrong information, the user must be given the possibility of deleting tags already found as well as adding new ones. This process should be non-manual, i.e. the user can simply use the mouse to mark the corresponding information and then click on a button which leads to the annotation of the marked text.

Annotation of text - manually

Some users prefer to annotate the text manually, i.e. enter the tags on a manual basis. The user should therefore be able to annotate the text directly without having to mark and click with the mouse.

Visualizing annotated text graphically

A visual annotation of text will only make sense if the user can also graphically see what he is doing. Therefore, depending on the enclosing tags, certain text fragments should be displayed using different colors.

Visualizing pure text

As the user should be able to directly annotate the text, he should also see the basic text with all its tags.

Changing the path to the ontologies

The ontologies, a user annotated, might be stored somewhere on his computer. Therefore, he must also be able to change the path to this location.

Changing similarity matching parameters

Many parameters have been shown in the concept for the similarity matching algorithm. All of them should easily be changeable, so that the user can evaluate the best parameter values for an exact similarity search.

Automatically save similarity ontology

If the user has annotated the text and the prototype inserted it into the ontology, the ontology must automatically be saved in the destination specified by the user. This way, the new ontology can be used for future matching processes.

Starting the similarity matching process

The user must be able to start the similarity matching process to evaluate the parameter values.

Visualizing the results

Once the process of comparing two ontologies is over, the results should be displayed immediately, i.e. the user should not have to wait for the similarity matching process to compare all ontologies, which were left out.

4.2 Used software components

After the requirements have been specified, the evaluation of potential components will be anticipated, because these components will already be incorporated into the static structure of the architecture. This thesis would not be possible without the use of components. Many necessary fields are simply too complex to cope with within half a year, therefore different components had to be chosen.

One big field is the NLP sector for the automatic annotation, where many different competitors exist. An evaluation of different tools is shown in 4.2.2. Another big part is the reasoning, which is treated in section 4.2.3. Section 4.2.4 finally shows which components have been used for the computation of the different similarity values.

4.2.1 General

A general cornerstone of this thesis was to settle the prototype on top of techniques which have been created in the Semantic Web area. Many of today's de-facto standards have been developed in this area, especially the Web Ontology Language (OWL [35]). The goal of OWL is to unify the former diverse area of ontology definition languages

in order to have one standardized, compatible and therefore highly interoperable language. It additionally supports the modelling of the ontology, described in chapter 3.2.3. Therefore OWL has been chosen as one cornerstone of the prototype.

The next thing to cope with was the decision, which programming language to use. Requirements for this choice were object orientation, high stability, a good performance and good tool support. The availability of the source of the framework was a nice to have, but not necessary. In the end, Java has been chosen. Just like OWL, it is a highly adapted programming language, which has evolved over the years. Further, many available components which are available on the Internet and are necessary for this prototype have been written using Java.

WordNet ([8]) is the most famous and freely available machine-readable dictionary. Many algorithms have been implemented using just WordNet as a dictionary. Due to this narrowing of choice, this prototype also relies on WordNet.

4.2.2 NLP

There were two requirements, which the component for the NLP core had to satisfy:

- Namefinding - This is necessary for detecting certain kinds of NF-requirements.
- Detection of coreferences - To refer information in a text to each other, coreferences of the text have to be detected.

Both are tasks which can of course be done by the user. Still, it was one requirement of this thesis to make this process at least semi-automatic. The first point (namefinding) is not critical. Mostly, all available frameworks in this area support this feature. Even detecting names via regular expressions yields sufficient results. The detection of coreferences, on the contrary, is a highly complex task, which involves a lot of research today. Many available frameworks do not support coreferences at all. Those who do support coreference detection, only find dependencies between nouns and pronouns. The ongoing scientific effort has been collected in the Computational Natural Language Learning (CoNLL) 2007 ([4]) which tries to push the scientists by putting them into direct competition. The different algorithms are being compared, using a training set which consists of nine different languages. Due to the complexity of this task, the following evaluation will compare several available frameworks with the main focus on their coreference detection capabilities.

Experimental parsers

As a result of CoNLL 2007, four different coreference detection systems have been identified which seemed to be available for download. One of them (CaboCha by Matsumoto and Kudo [17]) had to be eliminated from further evaluation due to its documentation in Japanese only.

- MaltParser - Nivre and Nilsson ([27]) created an algorithm which works as a shift-reduce parser ("shift" equals the shifting of an item onto a stack, whereas the "reduce" action simply pops the item from the stack. Sometimes, shift-reduce parsers are also referred to as "bottom-up" parsers). Their algorithm scored the second best average result on the CoNLL training set. It has been realized in MaltParser, using Linux and C / C++.
- DeSR - DeSR by Attardi and Ciaramita is a shift-reduce dependency parser which is based on the approach of Yamada and Matsumoto ([38]). They basically introduced a different set of rules as well as additional rules to handle non-projective dependency parsing in a single, deterministic step.
- MSTParser - The algorithm implemented in the MSTParser has been developed by McDonald, Lerman and Pereira ([24]). It basically uses a special scoring function to detect all dependencies between a head word y and another word x in a sentence. With these scores the dependency parsing problem is reduced to searching the highest scoring spanning tree in a directed graph (all words are nodes and the relation between words are edges). It has been implemented using Java 1.5 and is therefore platform independent.

In the end, none of these parsers could be fully evaluated due to different problems. MaltParser was platform dependent (Linux and C/C++) which posed problem with the basic requirement of this prototype being written in Java. The same problem occurred with DeSR which only existed in a closed source version for Linux. MSTParser was therefore the last known parser which seemed to be capable of solving the task. There are Java binaries as well. However, it didn't work, because it threw exceptions, as soon as either the training or the test-process was started. Furthermore, the requirements of the algorithm (1.8 GB of available heap space on a typical 32 Bit operating system or 8 GB on any 64 Bit OS) make it nearly impossible to use on any normal computer. Therefore these new parsers were not considered for further evaluation, although they looked promising in theory.

NLP frameworks

The NLP sector today contains several promising NLP frameworks, which are known to work quite well in certain aspects. Still, their coreference support is not as good as the experimental parsers promised in theory. Three of them have been evaluated:

- Gate - "Gate is the Eclipse of Natural Language Engineering, the Lucene of Information Extraction, a leading toolkit for Text Mining" - with this sentence, the user is greeted, when he visits the homepage of [Gate](#). Gate is a free open source

Java framework (GPL), consisting of a SDK and a graphical development environment which can be used to construct specific processing pipelines for different choices. It is best known for its information extraction capabilities.

- Lingpipe - [LingPipe](#) is a suite of Java libraries for the linguistic analysis of human language which sees its core capabilities in information extraction as well as data mining tools. There are free as well as commercial licences available, depending on the user type.
- OpenNLP - [OpenNLP](#) sees itself as "an organizational center for open source projects related to natural language processing". It therefore hosts several NLP projects. One of them is the "OpenNLP Tools" called framework which contains several components for sentence detection, tokenization etc. These components can be chained, suiting the user's needs to form a custom NLP processing pipeline. The framework is completely open source (LGPL) and written in Java.

The evaluation took place using always only one framework at a time (i.e. no mixing of components from different frameworks) and laying the focus on their coreference detection mechanisms. In simple text corpus examples, OpenNLP delivered equal or better results than the other two. Further, its smaller size (compared to Gate), 100% open source licence (compared to LingPipe) as well as a seemingly easier integration made it the primary choice for the work on this prototype.

4.2.3 Reasoning

As already mentioned, several inference machines have been developed in the semantic web area. [KAON2](#), [Pellet](#) and [Jena](#) have been picked out of a set of reasoning machines for a more explicit evaluation. These had to satisfy the following rudimentary requirements:

1. Available in a Java version
2. ABox reasoning
3. Rule and Query support
4. OWL-DL reasoning support

Table 4.1 shows a more complete overview of the different features, which the three of them support. All three of them possess an API which allows the creation and manipulation of loaded ontologies as well as saving and loading such ontologies. All three are based on completely different theoretical cornerstones. Especially the fast reasoning capabilities of KAON2 on large ABoxes are an interesting feature. Its algorithm

tries to reduce a knowledgebase to Prolog, so that known database algorithms can be applied. Pellets rule support is only of partial nature. KAON2 supports SWRL. Jena in turn has its own rule syntax. Based on experiences, gathered over several months,

<i>Feature</i>	<i>Jena</i>	<i>KAON2</i>	<i>Pellet</i>
API	Java/OS	Java	Java/OS
Ontology Manipulation API	Yes	Yes	Yes
Documentation	JavaDoc, Tutorials, Mailing list	JavaDoc, Source examples	Mailing list, JavaDoc,
Reasoning Alg	RETE	Datalog reduction	Tableaux
Reas. Support	TBox/ABox	ABox	TBox/ABox
Rule Support	Jena Standard	DL-safe subset of SWRL	Partial SWRL support
Query Support	SPARQL	SPARQL	SPARQL

Table 4.1: Reasoner Comparison

the final choice was Jena. It has a huge user basis and also offers a well documented API as well as many tutorials. Further, it offers an easily comprehensible rule syntax as well as a simple to extend rule engine. These are points which have proven to be of great value for the development of a prototype.

4.2.4 Similarity

The last components, which have been identified for reuse, were the algorithms for similarity measure. In both cases, not many components were available which implemented either the Jaro-Winkler metric (3.4.3) or the WordNet based similarity calculation by Lin (3.4.2). The latter has been implemented in the [Java WordNet Similarity](#) library, developed at the University of Sheffield. This is a native Java based conversion of the famous Perl-based [WordNet Similarity](#) package.

The Jaro-Winkler metric has been realized in the widely distributed [SecondString](#) library, an open source framework consisting of different string comparison algorithms. Both, the SecondString framework as well as the Java WordNet Similarity library, have been used for the implementation of the prototype.

4.3 Static Design

The design of an architecture comprises the determination of the basic structural framework of the system (see [33]). This implies the selection of specific components of the system. This section explains the static structure of the architecture, i.e. the classes and their relations between each other.

Basically, the SemRE prototype has been developed with the Model-View-Controller (MVC) pattern in mind (more details about this and the other patterns are shown in 4.3.5). When communication between the different levels of the MVC architecture is necessary (e.g. from the controller to the view level), the observer pattern has been used.

4.3.1 System overview

Figure 4.1 shows an overview of the architecture. Basically, it consists of six core packages: "gui" (view), "sysops" (shortcut for system operations, represents the core of the controller part), "nlp" (natural language processing), "similarity", "inbuilt" and "xml" (model). Basically, every "gui" component only accesses the core class "SystemOperations" (further referred to as SysOps), a singleton. This then initiates the necessary actions to fulfill the corresponding method call. The loading, saving and manipulation of text is done in the class "RequirementXMLModel". When a text has been loaded which is not XML compliant, i.e. it is not annotated, the SystemOperations will initiate an NLP process. The implemented NLP component has been hidden from the SysOps through the use of the abstract class "AbstractNLPComponent", which will facilitate the use of different NLP frameworks. If the user wants to initiate a similarity matching process, the SysOps will work on the abstract class "AbstractSimilarityProcess". This hides the complete reasoning and similarity matching logic from the SysOps which allows an easier implementation of other reasoners.

4.3.2 GUI

The GUI part of the architecture can be seen in figure 4.1. It basically consists of the main window "MainWindow" and two dialogues "ProgressDialog" and "SimilaritySearchDialog". The first one simply indicates the progress of the NLP process (it is notified through the IAutomaticAnnotationListener interface), whereas the second one allows the user to change several parameters, before starting the similarity matching process. It also puts out the results which have been gathered in the similarity matching process (notified by the ISimilarityListener interface). The "MainWindow" class holds a reference on two different Java TextAreas. One of them simply displays the direct XML text. This is done by using the "SourceTextRendering" class. It can be used to modify the text within the editor. The other one ("NormalTextRendering") colors the text, depending on its tags. These are eliminated from the output, so that the user is not confused with additional information. The "NormalTextRendering" class relies on the "Styles" class which associates a specific color to one certain tag.

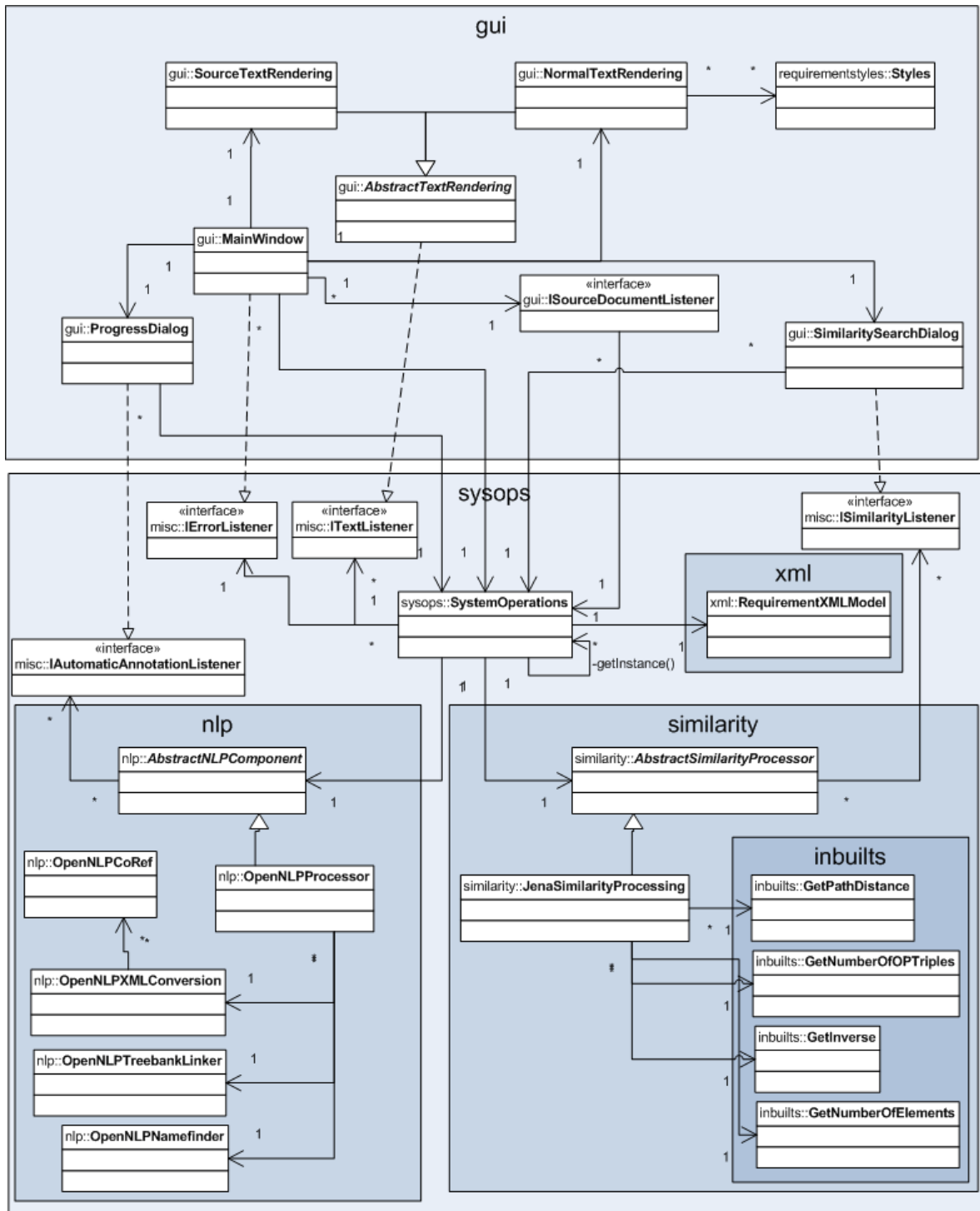


Figure 4.1: Overview of the prototype architecture

4.3.3 Inbuilts

The inbuilts package holds all inbuilts, which are necessary for the similarity matching process to work (see figure 4.2). All inbuilts have in common that they inherit from the Jena class "BaseBuiltIn". Below, the different inbuilts and their semantics are explained:

- `IsWordDistanceShortEnough` - calculates if the distance between two words is short enough (see 3.3.4).
- `IsWordDistanceRatioEnough` - calculates the ratio described in 3.3.4.
- `IsSimilarString` - calculates the similarity based on the similarity measure by Jaro-Winkler (3.4.3)
- `IsSimilarWord` - calculates the similarity based on the similarity measure by Lin (3.4.2)
- `IsSimilar` - combines both the `IsSimilarString` and `IsSimilarWord` measure to return one global value about the similarity of two strings (see 3.3.3).
- `GetInverse` - multiplies any given value with -1 to get the inverse of this number. This is necessary in the realization of the rules if two words are only traversable via the "inv_relatedTo" relation (see 3.3.4)
- `GetPathDistance` - calculates the length of a path between two words. It calls an implementation of the astar algorithm, whose distance measure to the neighbour word is always "1". This way, the algorithm behaves like Dijkstra (see 3.4.4).
- `AreSimilarRequirements` - calculates the similarity and satisfaction between two requirements. It uses the algorithm described in 3.4.1
- `AreSimilarNFRequirements` - inherits from `AreSimilarRequirements` and uses the same logic
- `AreSimilarArtefacts` - inherits from `AreSimilarRequirements` and uses the same logic
- `AreSimilarNFArtefacts` - inherits from `AreSimilarRequirements` and uses the same logic

All inbuilts are being registered at the corresponding reasoner. This way, they can be used in the rules (see 4.5.1). The AStar algorithm is split into five different classes: The class "AStar" holds the algorithm itself. It works on the class `AStarNode`, which is being compared, using the "NodeComparator" class. Every "AStarNode" has a

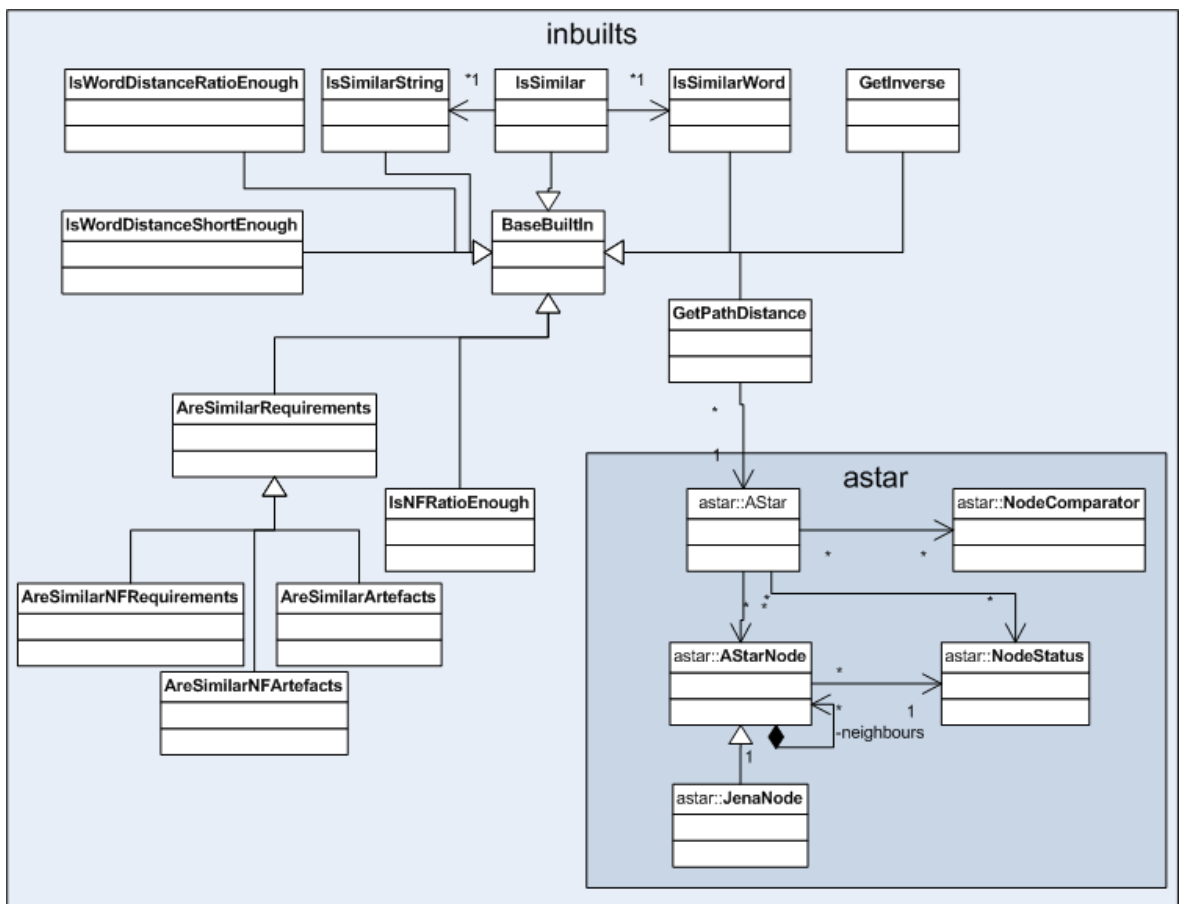


Figure 4.2: Overview of the inbuilts package

NodeStatus which states if the corresponding node has already been visited or not. "JenaNode" extends the "AStarNode" with its own logic, which allows the traversing of the Jena ontology.

4.3.4 XML

Basically, the annotation of the text is done using XML. Therefore, a simple structure for the creation and annotation of text has been designed (see 4.3). Every single tag, which is needed to annotate the document (see 3.2.1) has its own class, which in turn inherits from the super class "Tag". This class can hold an arbitrary amount of instances of the class "Attribute", which represents one XML attribute. The "RequirementXMLModel" class itself contains several instances of the different tags, depending on the annotation of the user.

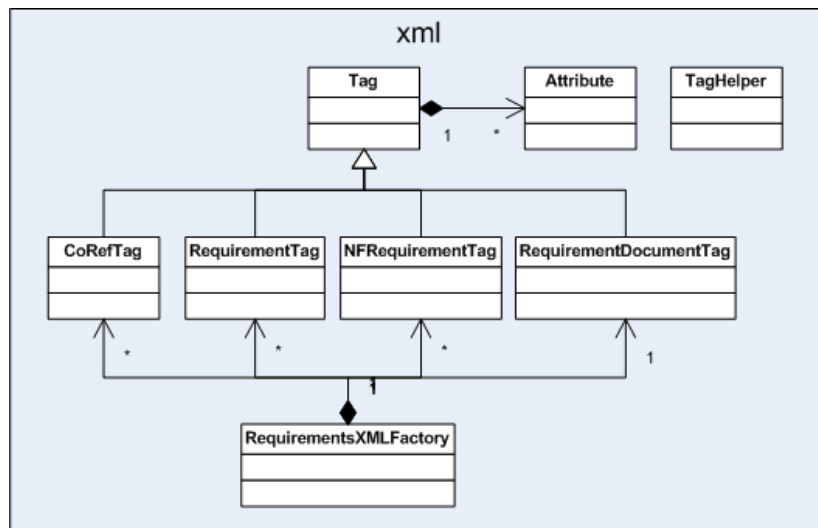


Figure 4.3: Overview of the xml package

4.3.5 Design Patterns

In the following, different patterns, used throughout the design of the prototype, are explained in short as well as where they have been used.

Model-View-Controller

The model-view-controller (MVC) paradigm is more an architectural style than a pattern. It strictly divides the whole architecture into three different levels:

1. The model which is responsible for the creation / manipulation etc. of data
2. The view which displays the relevant data to the user
3. The controller which holds the business logic of the application.

The strict separation of these three levels reduces the coupling between the different layers, which in turn increases the overall structure of the architecture. This again enhances the maintainability, the overall readability as well as the reuse of the code. Basically, the gui package and its contained classes represent the view part, whereas the whole systemoperations package can be seen as the controller. The data, which must be handled by the prototype, being neither too difficult nor too huge, is handled completely by the RequirementXMLModel.

Singleton

The singleton pattern is the solution to the problem, where only one instance of an object may exist during the runtime of a program. The advantage of this pattern is that the singleton class itself is totally controllable, i.e. the programmer using the singleton does not need to worry about how many instances of the object he is allowed to create / can create. This pattern has been used for the core SystemOperations class of the prototype.

Facade

The facade pattern is used whenever the programmer should not have to care about how the different methods needed must be called, i.e. it simplifies a number of probably complicated object interactions into one single interface. The SystemOperations class can be seen as a facade object, because it centrally catches all incoming user interactions and delegates them to the corresponding objects.

Abstract factory

The abstract factory pattern makes it easier for a programmer to implement different, but probably similar modules by integrating an additional interface layer. This way, the coupling between the program on one hand and the different components on the other one is kept as low as possible, facilitating the integration of additional components. This pattern has been used for both the nlp (AbstractNLPComponent is the super class of every to be integrated NLP framework) and similarity part (AbstractSimilarityProcessor). This way, further frameworks (either NLP or similarity) can be integrated into future steps to add new or better functionalities.

Observer

The observer pattern is used if one or more classes need to know about the state of another class. It allows for a high decoupling between the different layers of the architecture (MVC), using interfaces. It is used throughout the whole prototype to give feedback to the GUI. The corresponding interfaces for decoupling are `IErrorListener`, `ITextListener`, `IAutomaticAnnotationListener`, `ISimilarityListener` and `ISourceDocumentListener`.

4.4 Dynamic design

Every software system not only consists of a static structure, but also of a dynamic aspect. Therefore, the previous class diagram has been taken and reduced in its number of classes to the only relevant ones. These have been enriched with the most important public methods. Additionally, sequence diagrams will be used to give a better overview over the most important processes. Figure 4.4 shows the overview of the functional design.

4.4.1 Basic relevant functionality

Starting with the user's input, the loading of a new textfile is initialised. If the text file is not annotated, the systemoperations start the NLP process by calling the `processText(...)` method (for more information see 4.4.2). This sequence is shown in figure 4.5. During the NLP process, the NLP component updates registered instances of type `IAutomaticAnnotationListener` by calling the corresponding methods. When the whole process is finished, first the annotation listeners are notified, before the annotated text can be retrieved from the `AbstractNLPComponent` instance. This text is then again set in the local instance of the `RequirementXMLModel` and the known `ITextListener` instances are being notified that a change has happened to the existing text. Now the user can start working on the newly loaded text. If he wants to mark a text fragment or also clear the current text, he clicks the existing buttons, which in turn fire the corresponding methods in the `SystemOperations` singleton. This then first delegates these actions to the `RequirementXMLModel` instance, which modifies the text. After that, the `SystemOperations` again notify all registered `ITextListener` instances (figure 4.6).

If the user directly manipulates the text source (e.g., manually inserts new text), the `setText(...)` method in the `SystemOperations` is called. It forwards the new text to the `RequirementXMLModel`, which in turn checks, if the new text is XML compliant. If this is not the case, all registered `IErrorListeners` will be notified (see figure 4.7).

After having annotated the text, the user can simply save the annotated text in any

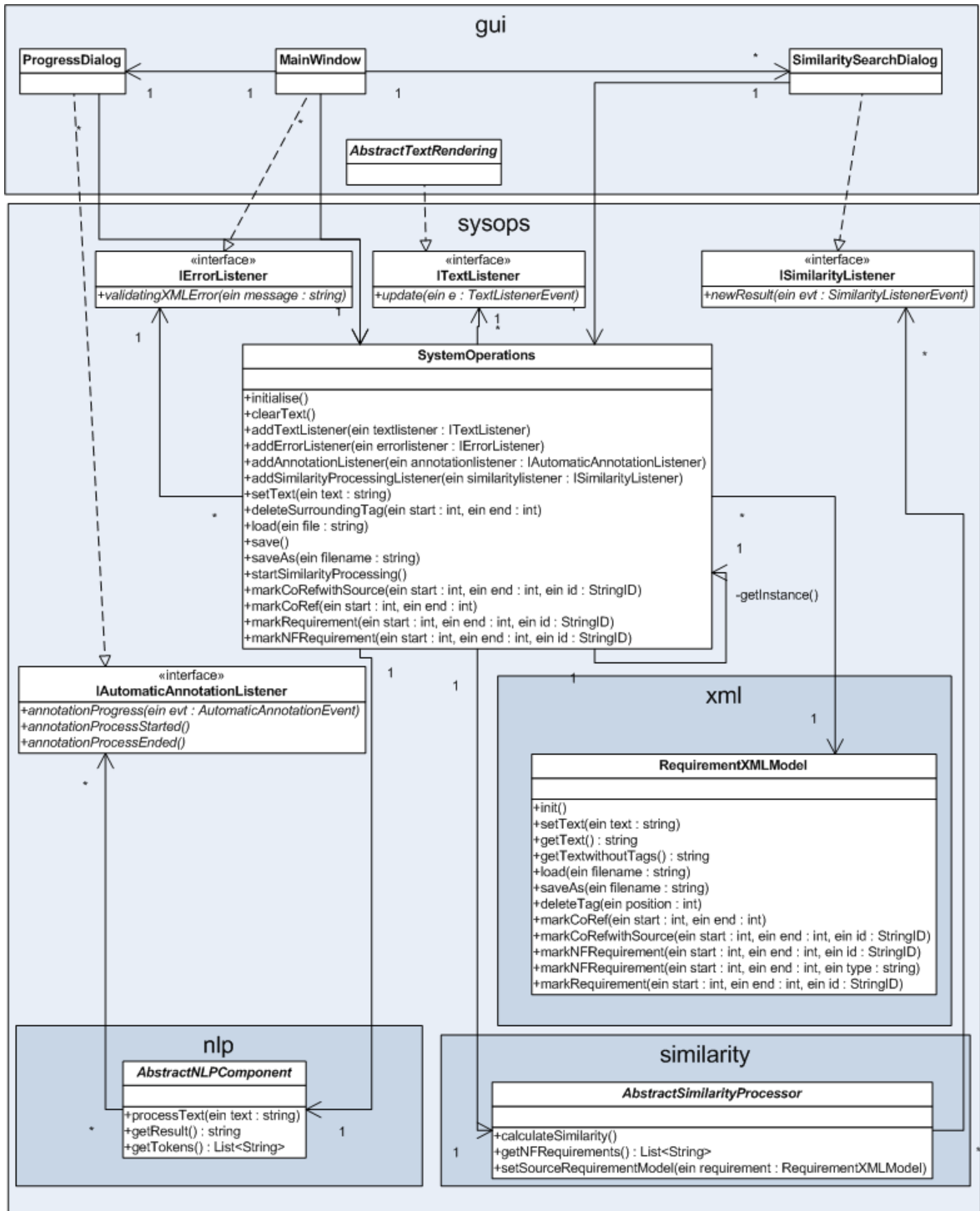


Figure 4.4: Overview of the functional design

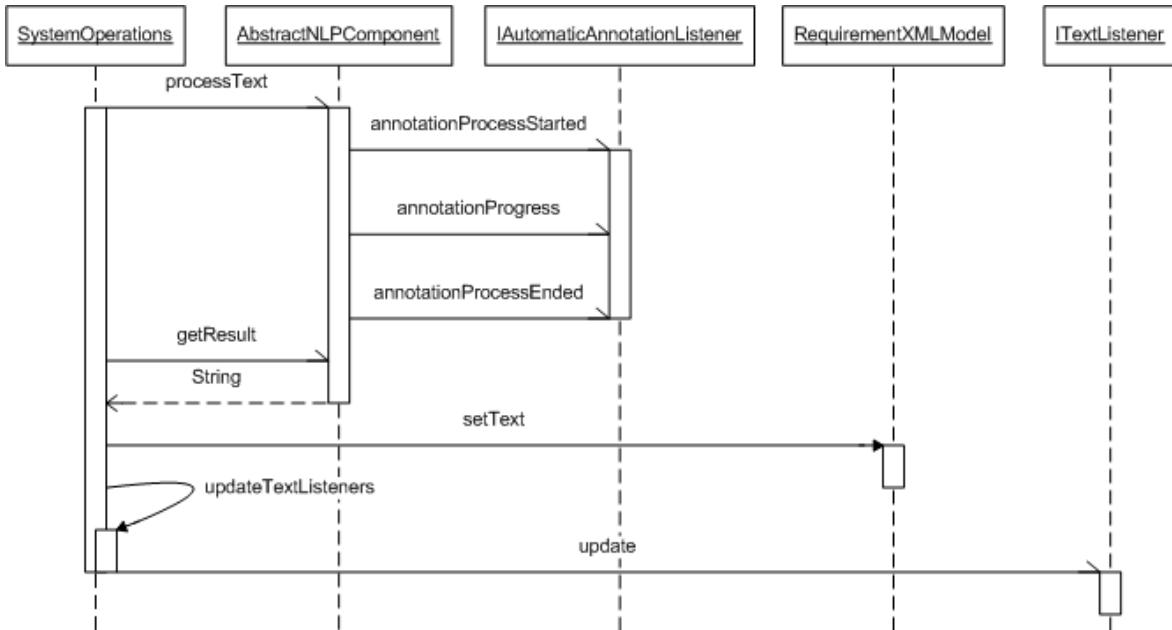


Figure 4.5: Sequence 1 - Processing of a newly loaded text file

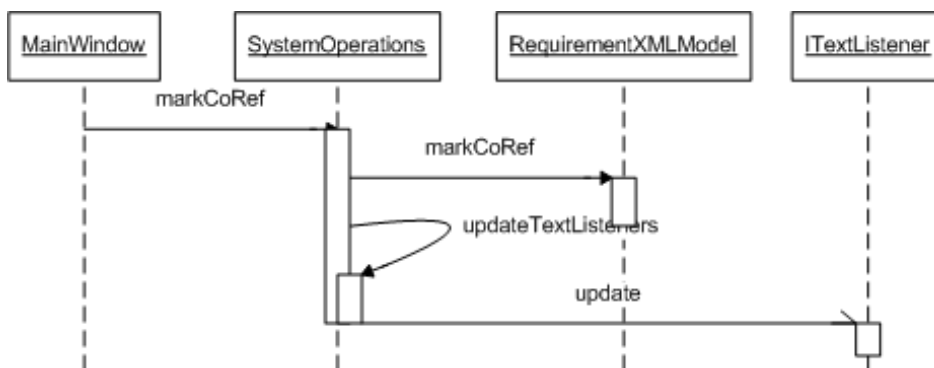


Figure 4.6: Sequence 2 - Exemplary sequence of text annotation

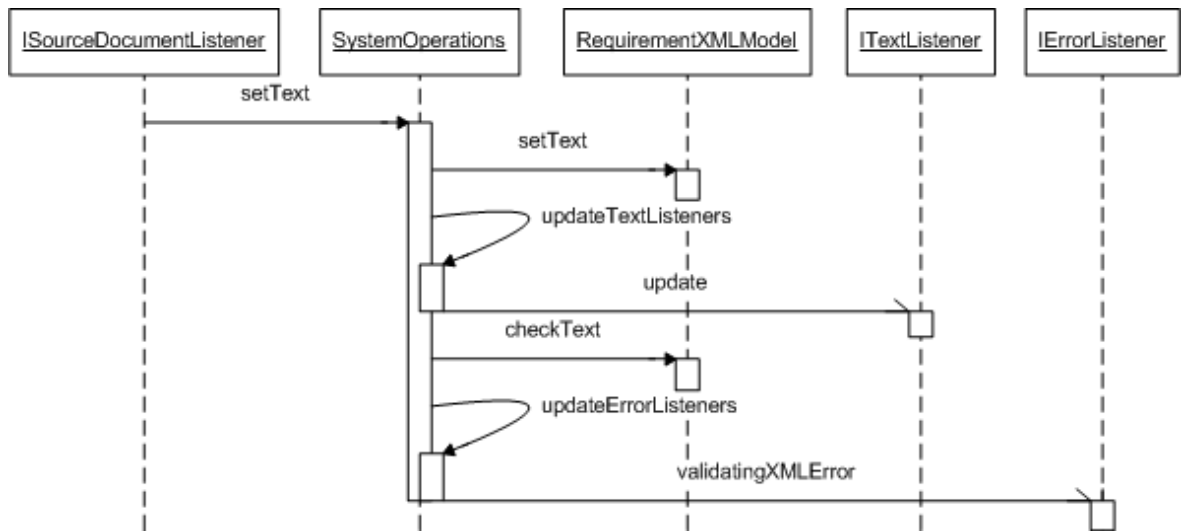


Figure 4.7: Sequence 3 - Failed manipulation of text source

file he wants. This is also forwarded to the RequirementXMLModel, which finally saves the text at the specified location.

If the user has finished annotating the text, he can start the similarity search (this sequence is shown in figure 4.8). In order to do so, he first opens the SimilaritySearch-

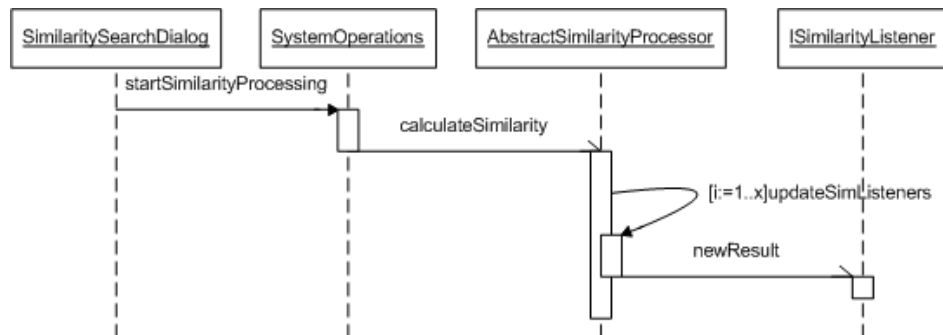


Figure 4.8: Sequence 4 - Starting of the similarity matching process

Dialog. After setting the different parameter values, he initiates the similarity search, which is then started by the SysOps in an asynchronously running thread. Every time two ontologies have been compared, the new results are sent to all registered ISimilarityListeners (for more information see 4.4.3). These can then take all important information and work on or display it to the user.

4.4.2 NLP functionality

The implemented NLP framework OpenNLP also uses a pipeline approach (see figure 4.9 for an overview of the relevant nlp methods). After getting new, not annotated text, the system starts by first splitting the text into its single sentences. When this process is finished, it continues with the tokenization process. It takes every single sentence and splits it into its tokens. The resulting list of tokens is then given to the treebank parser (a treebank is a text corpus, in which each sentence has been annotated with a syntactic structure), which, according to the syntactic position of every word, assigns a tag to every single token which states its semantic meaning (e.g., if it is a pronoun, a noun, an adverb etc.). Based on these pieces of information, a namefinding process is started, which tries to identify names, according to a set of pre-trained models. After all these steps, the final phase in the NLP chain is the detection of coreferences. Next, the annotated results from the OpenNLP framework are taken and given to the OpenNLPXMLConversion, which inserts the newly found information as XML tags into the original text. This is done by first iterating over the result data structure and putting all coreferences found into an internal hashtable, wrapping them in the OpenNLPCoRef class. Then, a recursively working method works on the tree-like parsing results (every syntactical information, either of the treebank, the namefinder or the coreference module, is one node in the tree). If a node in the tree is either a coreference or a name, a corresponding tag will be inserted into the original text, using the RequirementXMLModel.

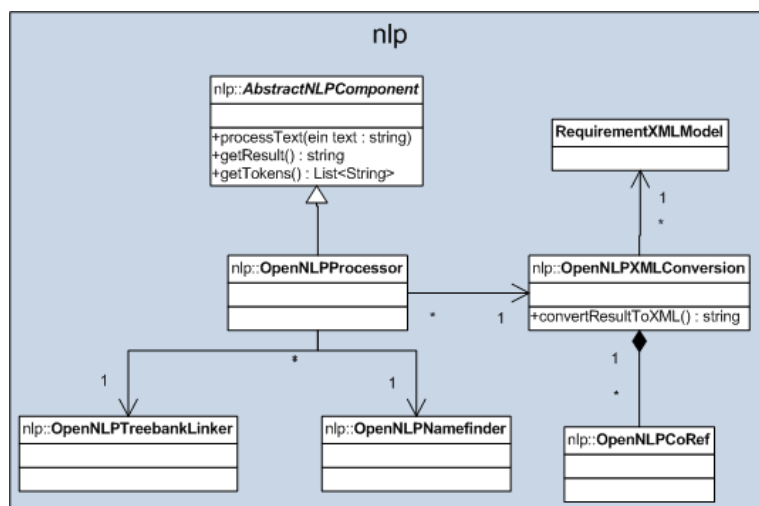


Figure 4.9: Overview of the NLP functionality design

4.4.3 Similarity functionality

The annotated text is given to the corresponding similarity instance, which in this case works on top of Jena. First, the new text needs to be inserted into the ontology. This is done by retrieving the text from the RequirementXMLModel. It is then split by its tags. The text fragments between the tags are given to the NLP component for tokenization. Every token is a possible element for insertion into the ontology, i.e. an individual of the type "Word". Then all available stopwords (i.e. words, which carry no important information like "a", "an", "about" etc.) are removed from the tokens list. Finally, the tags and tokens are combined in a single list. Based on a stack system, which holds the available tags, the different tokens are inserted into the ontology, following some conditions:

- Every word is related to its predecessor word
- Every word is related to a requirement. If the user forgot to mark a word with a requirement, this word is related to a standard requirement.
- Every item is given a unique namespace.

This way, the ontology is constructed and finally saved in the user specified location. With the ontology of the text existing, the similarity matching process is started. Therefore the rule files for every phase are loaded. Every ontology of the user specified location is loaded and joined to the newly created one. Based on this "super" ontology the inference starts. If no NF matching could be detected (see 3.3.2), the inference process for this ontology is cancelled and restarted with a new ontology from the location. After the whole process has finished, specific information is collected from the inferred ontology. All registered ISimilarityListeners are then notified that a new result exists.

4.5 Implementation

The implementation finally means the concrete realization of the developed design. However, this chapter will not show Java code, but focus on the description of implementation specific details, because some parts of the concept could not be realized one to one. Further, this chapter gives an overview of the realization of the graphical user interface.

4.5.1 Jena and its rules

As the concept in 3.3 showed, the basic similarity matching process consists of four different main phases. However, due to the use of non-monotonic and non-logical

inbuilts (e.g., the counting of elements in the ontology), some of these four phases (starting with phase 1 (3.3.2)) had to be split further. To give a better insight into the implementation, the different rule layers will be explained:

1. Basic additional inference results are added to the ontology as well as the "isSimilarWord" relation for NF requirements.
2. The similarity of the NF-requirements (represented through the "isSimilarNFRequirement" relation) is computed. The corresponding inbuilt "AreSimilarNFRequirements" depends on "isSimilarWord" and can therefore only be executed, after all possible "isSimilarWord" relations have been identified.
3. In this phase, the similarity of artefacts regarding their matching NF-requirements is calculated. This is done among others by counting the number of "isSimilarNFRequirement" relations.
4. This phase will only be executed, if after phase three a new relation of type "haveMatchingNFRequirements" can be detected between both artefacts. If this is the case, "isSimilarWord" relations will be inferred for every normal requirement.
5. Based on the results of phase four, the structural matching will continue. This one adds the "isStructuralMatch" edges to the ontology.
6. Now the similarity of the normal requirements is calculated, which incorporates the counting of "isStructuralMatch" relations. New "isSimilarRequirement" edges are inserted into the ontology.
7. The last step is the creation of "isSimilarArtifact" objectproperties, which requires the counting of results from phase six.

The first three layers are responsible for the process, described in 3.3.2, whereas layer four and five correspond to 3.3.3 and 3.3.4. Layer six and seven implement the theory of phase four (3.3.5).

4.5.2 Concrete tags and file structure

The necessary tags have already been shown in section 3.2.1. The concrete realization of these tags has been done using XML. This is best explained by simply giving a completely annotated file structure:

```

<requirementdocument><requirement id="1">The music <coref id="3">
player</coref> should support the playback of mp3 files and run on
<nfrequirement type="OperatingSystem" id="1">Windows XP
</nfrequirement >. </requirement><requirement id="2">
```

```
Additionally , <coref source="3" id="4">it</coref> should be easy  
to install and should have a short start time.</requirement>  
</requirementdocument>
```

The file starts, XML-like, with the root element, in this case "requirementdocument". A requirement is marked with the "requirement" tag. Note that every tag except the "requirementdocument" has an "id" attribute, which identifies the corresponding tag. NF-requirements are marked, using the "nfrequirement" tag, which additionally contains a "type" attribute. This states the type of the corresponding NF-requirement and can only be one of the NF-requirement elements, shown in the ontology structure (see 3.2.3).

A coreference is marked by a "coref" tag. If a coreference refers to another coref, it additionally contains a "source" attribute, which refers to the coreference id of the anchor "coref" tag.

4.5.3 Graphical User Interface

For easier interaction with the user and to demonstrate an easy usage of the whole concept, a graphical user interface has been implemented. Still, a user must not think that a GUI is easy to use, because it does not suit his sense of taste.

The design of the GUI follows in this case the traditional design of windows programs, as can be seen in figure 4.10. This is the window the user gets to see, if he freshly starts the program. The design is basically split into three different parts: The menu in the upper area of the window, the annotation tabs on the left side and the two different text-output tabs on the right side.

The menu allows the user to load or save a file as well as to quit the program. Further, the user can start the similarity matching dialogue (see figure 4.11). When a new file has been loaded, its text is highlighted, according to its tag information. Requirements are marked in green, coreferences with blue and NF-requirements with red. Basic, not annotated areas of a text, are displayed using simple black color. The user can now select text with the mouse and then use the tabs on the left side to add new tags. This is done by simply selecting for example a type in the NF-requirement tab and clicking on the NF-requirement button. This way, a new NF-requirement tag is inserted around the selected text area. At the bottom of the left side, the user can specify the name of the artefact. If the user wants to annotate the source directly, he simply switches from the "Normal Text Rendering" to the "Source Text Rendering" tab. This does not support text-highlighting, but allows for the direct editing of text.

When the user has finished, he can open the similarity search dialogue (see 4.11). It gives the user the possibility to change the location to the ontologies. After that, he can change different parameter values (which have been described in 3.3 and will be wrapped up in 5.1.1) and start the similarity matching process. Every new artefact

result is shown in the first table. If the user chooses one entry from this table, the text of both artefacts will be shown in the corresponding textareas. Additionally, all matching requirements are shown in the second table. If the user selects an item from the requirement table, its structural matching words will be displayed in the last table.

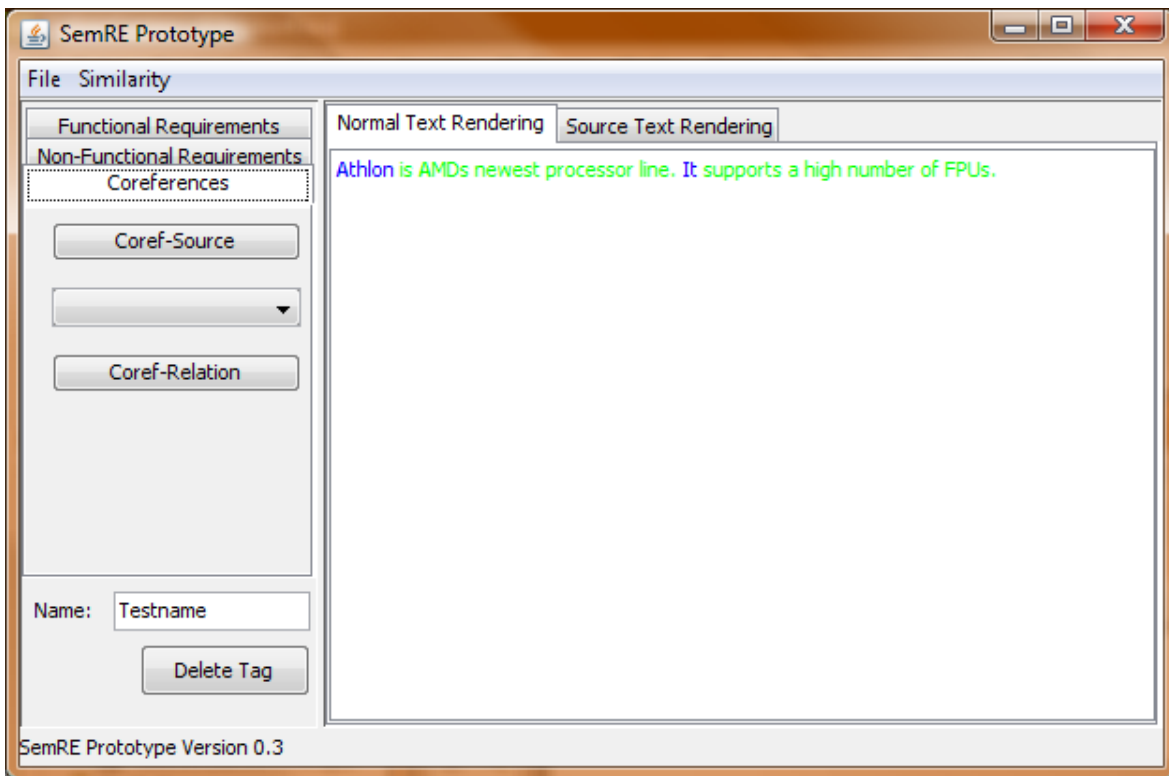


Figure 4.10: The main window of the prototype

Similarity Measurement

Folder to components:

NF Requirements Similarity Threshold 0.01

NF Artefact similarity Threshold 0.01

String Similarity Threshold 0.8

Word Similarity Threshold 0.8

Max. word distance 3

Max. word distance ratio normal 0.8

Max. word distance ratio inverse 0.8

Requirement similarity threshold 0.25

Artefact similarity threshold 0.01

Source artefact	Dest artefact	Satisfaction	Similarity
ann_microsoftsql	ann_requirementdocument	0.0	0.0
ann_requirementdocument	ann_microsoftsql	0.0	0.0
ann_requirementdocument	ann_mysql	0.5	0.23076923
ann_mysql	ann_requirementdocument	0.15	0.23076923
ann_requirementdocument	ann_secondstring	0.0	0.0

Source Artefact Text

Destination Artefact Text

process should be reasoning based, using an oncology tbox, which is defined, using OWL-DL. Based on different measures, for one the similariy between two strings should be computed, using the Jaro-Winkler measure. The semantic similarity should be calculated with the Lin metric. The lexical database, which is used by the Lin algorithm, can be freely considered. All components should be accessible, using Java.

engines; it has savepoints with innodb. it supports SSL. it support Query caching. It allows Sub-SELECTs (i.e. nested SELECTs). It allows for replication with one master per slave, many slaves per master, no automatic support for multiple masters per slave. It has Full-text indexing and searching using MyISAM engine. It has an embedded database library. It is ACID compliant using the InnoDB, BDB and Cluster engines.

Source requirement	Dest requirement	Satisfaction	Similarity
ann_mysql->Requirement_92	ann_requirementdocument-...	0.0	0.0
ann_requirementdocument-...	ann_mysql->NFRrequiremen...	0.0	0.0
ann_mysql->Requirement_44	ann_requirementdocument-...	0.0	0.0
ann_requirementdocument-...	ann_mysql->Requirement_53	0.0	0.0
ann_mysql->Requirement_151	ann_requirementdocument-...	0.0	0.0
ann_mysql->Requirement_135	ann_requirementdocument-...	0.0	0.0
ann mysql->NFRrequiremen...	ann requirementdocument-...	0.0	0.0

Source Word	Dest Wort	Similarity	Comparison method
-------------	-----------	------------	-------------------

Figure 4.11: The similarity window

Chapter 5

Evaluation

The evaluation of the prototype developed was done using an example set of requirements, which have been either written down by hand or taken from existing components. All of them have been annotated using the available possibilities of the prototype.

The main goal of this chapter is the evaluation of the matching process, whereas the NLP part of the prototype will be left out. First, the evaluation criteria as well as the different available parameters (which can be seen on figure 4.11) are explained in 5.1. As currently no basic parameter value set exists, one will be evaluated in section 5.2, using a small set of components. Based on the newly found values, section 5.3 presents the evaluation of a bigger set of components and shows how they match a sample requirement document.

5.1 Evaluation criteria

As already mentioned, one of the main goals of this section is to find the most suitable parameter values for the matching process. Therefore, the different parameters and their semantics will be explained in 5.1.1. The parameters which are of relevance for the evaluation of the process are shown in 5.1.2.

5.1.1 Parameters

During chapter 3 many parameters have been introduced which directly influence the potential matching result between two ontologies. All of these parameter values can be changed by the user, as it is shown in figure 4.11. In the following, all parameters and their semantics are explained.

- NF Requirements Similarity Threshold - This parameter states, how many words of two NF-requirements must have an "isSimilarWord" relation for the require-

ments to be similar, e.g., if every NF-requirement contains four words, of which two words have an "isSimilarWord" relation to two words of the other NF requirements and the threshold is 0.5, both NF-requirements will be similar.

- NF Artefact Similarity Threshold - Defines how many NF-requirements of two artefacts must be similar for the artefacts to have similar NF-requirements. The semantics are the same as NF requirements similarity threshold.
- String Similarity Threshold - This is the threshold which the syntactic string comparison needs to exceed for two words being similar.
- Word Similarity Threshold - This is the threshold which the WordNet based string comparison needs to exceed for two words being similar.
- Max. word distance - Defines the maximal number of steps which are allowed to get from one word from to another.
- Max. word distance ratio normal - Defines the ratio which the distances between two word pairs must exceed.
- Max. word distance ratio inverse - Defines the ratio which the backwards distance between two word pairs must exceed
- Requirement similarity threshold - This parameter states how many words of two requirements must have a "isSimilarWord" relation for the requirements to be similar. The semantics are the same as NF requirements similarity threshold.
- Artefact similarity threshold - Defines how many requirements of two artefacts must be similar for the artefacts to be similar. The semantics are the same as NF requirements similarity threshold.

Based on the comparison of three components against each other (one combination matches, the others do not), a basic evaluation of these parameters takes place with the aim of finding a basic but reasonable parameter set for making a more detailed evaluation of the algorithm. Not all of these parameters are of high importance to the evaluation process. Especially the NF-requirement as well as (NF-) artefact similarity thresholds will not be considered any further in this chapter. The cause for not regarding these three similarity parameters is that the approach used in this thesis is a bottom-to-top approach, i.e. based on the results gathered at the bottom the similarity is computed. The similarity in this case is mostly dependent on the syntactic, semantic and structural similarity between word pairs, which are calculated using the string- and word similarity threshold as well as the max. word distance parameters. The NF-parameters are not evaluated at all because they are highly dependent on what the user really wants, i.e. how important the non-functional part is to the user. Also

the normal artefact similarity threshold will not be regarded, because its only influence is if an "isSimilarArtefact"-edge should be inserted between two artefacts or not - this will only be important for finding already matched artefact pairs in a repository.

5.1.2 Comparison

The direct comparison of two ontologies is measured by the satisfaction and the similarity between their requirements and the artefacts themselves. Their calculation has been described in 3.4.1. More concrete, based on the following conditions, the evaluation takes place.

1. The higher the similarity value between two items, the better one matches the other.
2. The higher the satisfaction of one item towards another, the more information seems to be matching.

A requirement document may not always match exactly one complete other component. More probably, only some requirements of the whole document match the requirements of other components. Therefore, based on the satisfaction, components are selected and then compared by their similarity values. Only results of components, which have been evaluated completely, i.e. they have matching NF-requirements, so that an overall similarity value could be computed, will be put into one of the categories above.

As the algorithm uses a bottom-top approach, it is necessary for the parameters which control the results of the lowest level to provide good and reasonable results. Therefore, the evaluation of the parameters will be divided into three different levels. The levels which are to be evaluated and their corresponding parameters are shown in the enumeration below:

1. Syntactic / semantic matching: String / Word Similarity Threshold
2. Structural matching: Max. word distance, Max. word distance ratio normal / inverse
3. Requirement / artefact matching: (NF-) Requirements / Artefact Similarity Threshold

5.2 Basic parameter values

As already mentioned, it is necessary for the further evaluation to gather a basic, but 'working' parameter set. Therefore, three simple component descriptions have been taken: The first one is a description of a standard musicplayer (mp1), and consists

of a total of 61 words (note: This is the number of words, which have been inserted into the ontology). The second one is a description of MySQL (mysql) and contains 68 relevant words. Both descriptions are matched against a simple user request for a music player (mp2). It holds a total of 27 words.

5.2.1 Syntactic / semantic parameter evaluation

To adjust the parameters for the syntactic and semantic similarity, several processes with the parameters from table 5.1 have been started. As can easily be seen, all parameters except for the string and word similarity thresholds (marked with an 'x') have been set to the lowest possible value. The cause for not setting the word- and string values to 0.01 is, that with 0.01 so many wrong similarities would be found that the rest of the matching would take several hours to complete, even with these small examples. Therefore, both will be set to 0.5 (50%) and then steadily increased in steps of 0.05 (5%) up to 1 (100%).

Still, with these parameter values, there can be no real indication on how similar one artefact description is towards another. However, this is not yet the intended goal. Primarily, only basic string- and word similarity thresholds will be evaluated. Based on the 'isSimilarWord' relations between word tuples, all results have been counted, separated by syntactic and semantic comparison as well as if a human would see a similarity between both words or not. These results can be seen in figures 5.1 and

<i>Parameter</i>	<i>Value</i>
NF Requirements Similarity Threshold	0.01
NF Artefact Similarity Threshold	0.01
String Similarity Threshold	x
Word Similarity Threshold	x
Max. word distance	1
Max. word distance ratio normal	0.01
Max. word distance ratio inverse	0.01
Requirement similarity threshold	0.01
Artefact similarity threshold	0.01

Table 5.1: Basic syntactic / semantic parameter set

5.2. First of all, the number of results is highly different between the syntactic and the semantic (WordNet based) matching process. The cause for this is that the WordNet comparison requires both words to be of the same type (e.g., both verbs), whereas this is of no matter to the syntactic comparison. In both diagrams the blue bars indicate the results which based on human judgement are correct. The red ones represent wrong matches. In figure 5.1, there is a large number of wrong results at 50%, continuously decreasing while going towards 80%, from where on the number of correct hits

increases. Therefore, the string similarity threshold parameter will be fixed at 80% for the ongoing evaluation. Regarding the semantic matching process in figure 5.2, it shows a similar picture. From 50% to 80% a high number of mismatches can be seen, whereas the number of correct results increases starting with 80%. Still, the "break" is not as sharp as it has been seen at the syntactic matching. The cause for the line not falling continuously from 50% to 80% may be due to the relatively small number of result items. Due to the "break" at around 80% (more correct than wrong results), the semantic similarity threshold parameter will also be fixed at 80%.

One additional advantage about a high similarity threshold (besides having good results) is a largely decreased number of items in the result set. This in turn greatly boosts the overall performance of the system, because it does not have to make as many calculations as with the lower threshold value. Figure 5.3 gives an overview of the time used for the comparison of both similarity thresholds (0.5 and 0.8, all other parameters have been left as denoted in table 5.1). Note, that these times have been taken using a complete matching pass between mp2 and mp1 as well as mp2 and mysql on a Core 2 Duo 3.0 GHz system with 3 GB RAM. The difference between both parameter values is tremendous. It takes nearly 45 minutes to complete the whole matching process with the value 0.5, whereas just 2.5 minutes are necessary for 0.8.

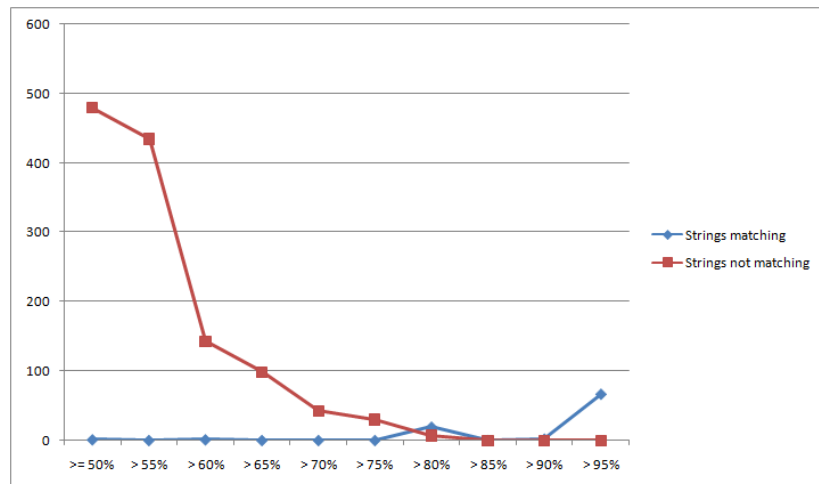


Figure 5.1: Syntactic word matching

5.2.2 Structural parameter evaluation

The structural matching process can be influenced with three different parameters: Max. word distance, Max. word distance ratio and Max. word distance ratio inverse. The most influencing of these three is the first one which defines the max. distance which is allowed between two words. The other two parameters simply help to refine the

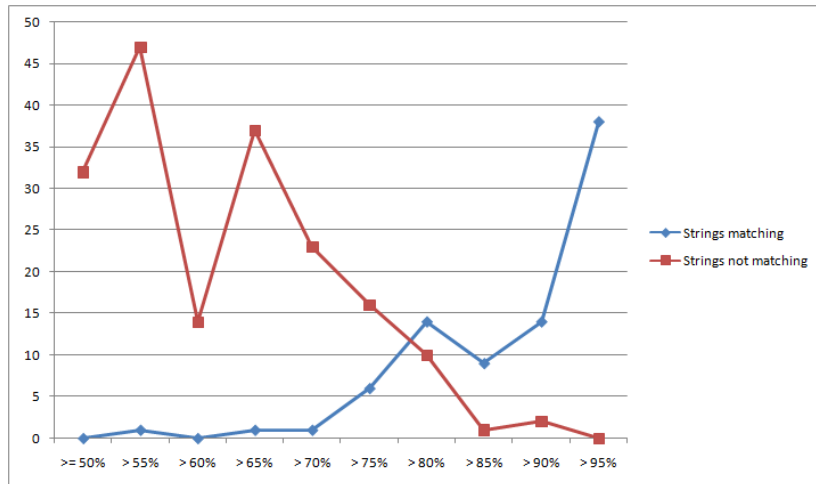


Figure 5.2: Semantic word matching

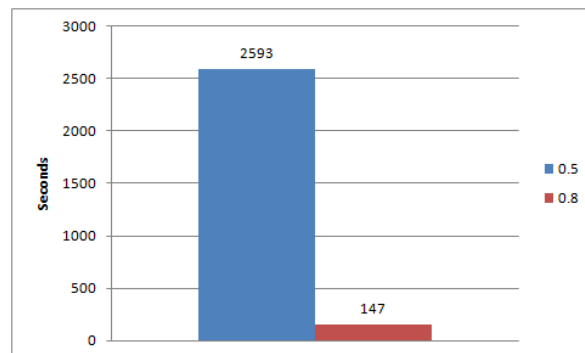


Figure 5.3: Performance comparison of similarity threshold parameters

ratio, also described in section 3.3.4. In short, the first parameter helps to specify that there is a similar context structure existing between both artefact descriptions, whereas the other two just further refine the results. Based on the same examples from the last section, the first parameter (again marked by an 'x1' in table 5.2) will be evaluated for values between one and ten. 'x2' and 'x3' will be left at 0.1 in the first instance. The results can be seen in figure 5.4. The blue line indicates the matches found between both similar components, the red and green one between the dissimilar ones. Starting with the lowest possible value 1, both music player descriptions already have a large set of overlaps, whereas there are none between mp2 and the database. Also, only a few can be found between mp1 and MySQL. By increasing the value more overlaps are detected for all comparison lines, until a certain maximum number is reached (this of course depends on the number of similar words found in the previous step). The optimum value seems to be at 3 in this case because the blue line (which represents the only two similar components) has nearly reached its maximum number of overlaps, whereas the other two have not. This makes sense because similar information also has a closer regional relation. Further, the distance between all three lines is the largest one at this point which will also help to differentiate the final similarity values.

One thing that meets the eye is the parallel course of all three lines. Especially both comparisons which have been made with MP2 are very soon 'satisfied', i.e. no further results are found. This of course also depends on the size of both text corpora: The larger they are, the more structural information can be found by increasing the path distance value. This can be seen by looking at the green bar: Both textual descriptions have more than 60 words and the number of possible matches still increases from 3 on. However, this value also converges at an unknown maximum, because there cannot be an unlimited number of possible similarities, i.e. nearly all possible structural similarities between both ontologies have been found.

Still there is doubt that a much larger context area will automatically help to produce better results. For now, 3 is the value of choice and will be used in the rest of the ongoing evaluation.

Note, that a higher number of structural matches does not automatically mean a higher similarity. It is the ratio between structural similarities found and words not "used", which leads to the final similarity value (see 3.3.5).

With a set value for the max word distance, both ratios will now be evaluated. Beginning with 'x2' at 0.1 (10%), it will be increased up to 1 (100%) in steps of 0.1, whereas 'x3' will be left at 0.1. Evaluating 'x3' is done under the same conditions as 'x2'. Because the max. path distance has been chosen to be 3 the values do only change in three steps ($\frac{1}{3}$, $\frac{2}{3}$ and $\frac{3}{3}$). This can be seen in all three figures (5.5, 5.6 and 5.7). Figures 5.5 and 5.6 (the percentage numbers in the brackets show the total loss of structural matches) indicate that changing a single parameter does not have a huge impact on the number of structural matches. The cause for this is that one normal structural match

<i>Parameter</i>	<i>Value</i>
NF Requirements Similarity Threshold	0.01
NF Artefact Similarity Threshold	0.01
String Similarity Threshold	0.8
Word Similarity Threshold	0.8
Max. word distance	x1
Max. word distance ratio normal	x2
Max. word distance ratio inverse	x3
Requirement similarity threshold	0.01
Artefact similarity threshold	0.01

Table 5.2: Basic structural parameter set

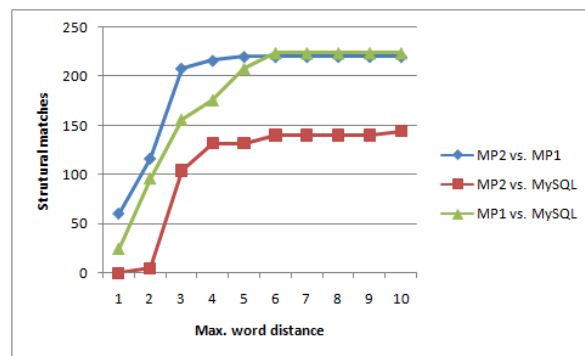


Figure 5.4: Comparison of max. path distance parameter

can often be compensated by its inverse relation ('relatedTo' and 'inv_relatedTo'). This fact is also substantiated by figure 5.7: The results in this picture have been created by changing both parameters synchronously - A larger impact on the number of items is the consequence.

Based on figure 5.7, a high value for both ratio parameters seems to be a good choice, as it has a higher impact on the dissimilar component pairs than on the similar ones. This also underlines the fact that similar component pairs more often have a similar structure of information than dissimilar ones. Therefore, a value of 80% is taken for the rest of the evaluation.

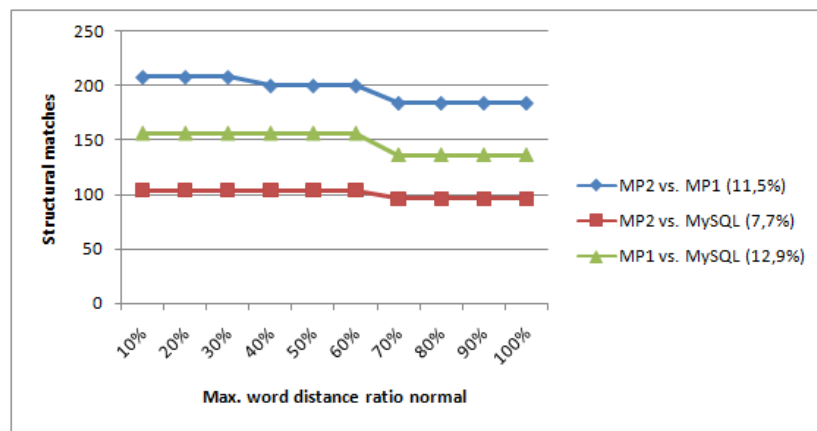


Figure 5.5: Comparison of max. path distance ratio parameter

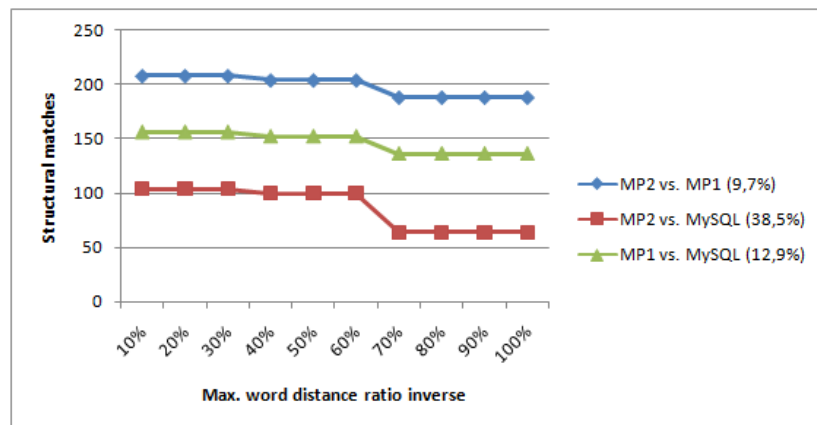


Figure 5.6: Comparison of max. path distance ratio inverse parameter

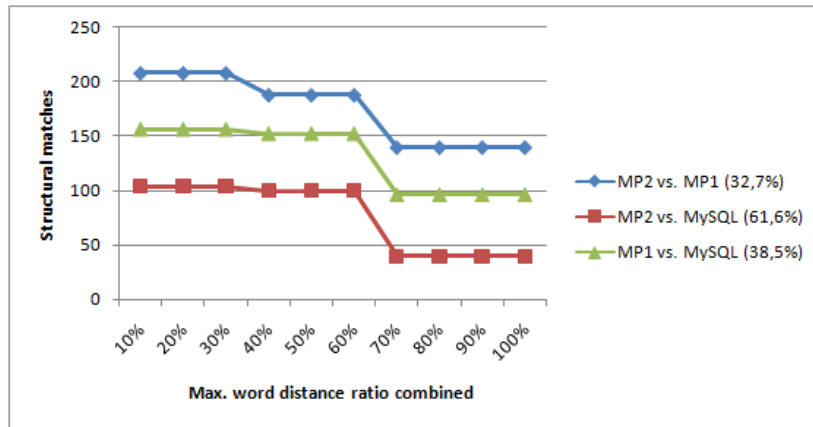


Figure 5.7: Comparison of max. path distance ratio combined parameter

5.2.3 Requirement parameter evaluation

Finally, the last parameter to be evaluated will directly influence the similarity threshold. The requirement similarity threshold defines which threshold of similar to dissimilar words must be transcended for two requirements to be similar (see 3.3.5 for a more detailed explanation). The sample set is again the same as in the previous section.

The evaluation will start with a value of 0.01 for x_1 . This way, all possible results can be gathered and simply compared in a chart (see 5.8). As it can easily be seen, both dissimilar component pairs (MP2 vs. MySQL, MP1 vs. MySQL) have between 80% and 90% requirements with a similarity ranging from 0% to 10%. Both music players, however, share only 40% of their requirements with a similarity lower or equal to 10%. Additionally, the blue line more often shows that there are requirements with a higher similarity, in contrast to the dissimilar component pairs. Looking at the chart it will be sufficient for the ongoing evaluation to fix the requirement similarity threshold at 30%. This will filter the biggest part of unnecessary and not matching requirements and still leave enough correct requirement matches, regarding the similar component pairs.

One thing which is not so easy to detect in the chart is that all three lines have a requirement with a similarity between 90% and 100%. This is, in all three cases, the non functional requirement which is also included in the similarity calculation.

With all these parameter settings evaluated, the similarity results from figure 5.9 have been calculated for the basic component set. "Satisfaction \rightarrow " states how much of, e.g., MP2 is satisfied by MP1, whereas "Satisfaction \leftarrow " defines the same for how much of MP1 is satisfied by MP2. As can easily be seen, MP2 has a much higher similarity and satisfaction to MP1 as to MySQL. The same goes for MP1 vs. MySQL, which have a slightly higher similarity than MP2 vs. MySQL, but still a very low one, compared to MP2 against MP1.

<i>Parameter</i>	<i>Value</i>
NF Requirements Similarity Threshold	0.01
NF Artefact Similarity Threshold	0.01
String Similarity Threshold	0.8
Word Similarity Threshold	0.8
Max. word distance	3
Max. word distance ratio normal	0.8
Max. word distance ratio inverse	0.8
Requirement similarity threshold	x1
Artefact similarity threshold	0.1

Table 5.3: Basic requirement parameter set

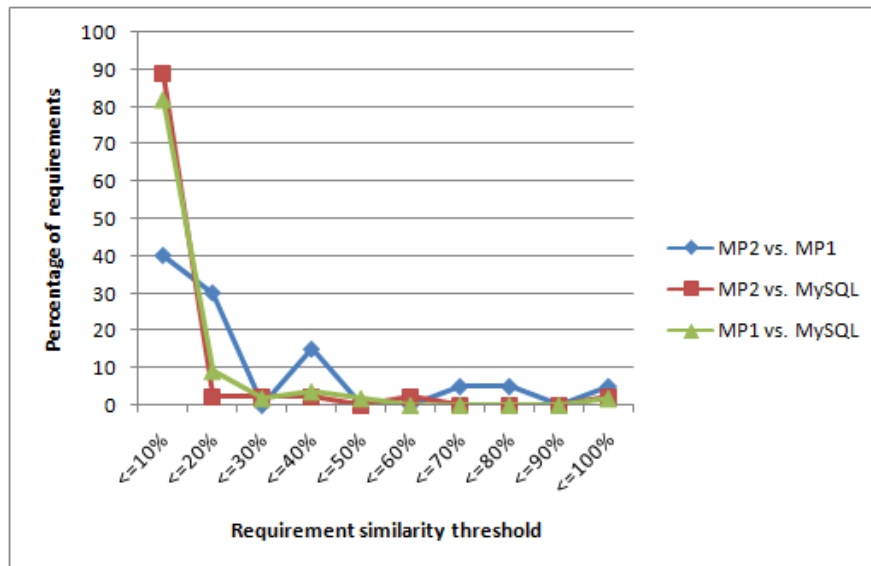


Figure 5.8: Overview of all found requirement pairs and their similarity

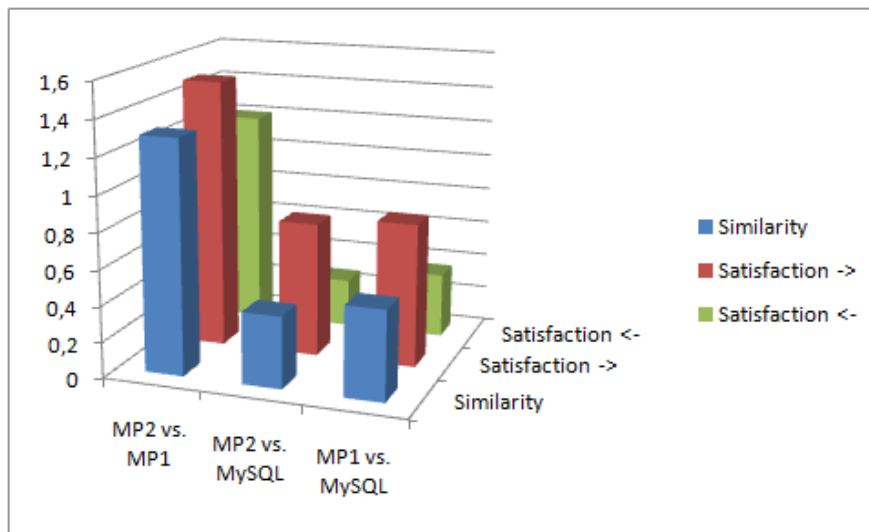


Figure 5.9: Similarity between basic similarity set components

5.3 Requirement document evaluation

The last section showed the way how a potential first parameter set which has still to be refined has been found. This set will now be used to make an evaluation of a sample component description set which is matched against a simple requirement document.

5.3.1 Testdata

Because the tool and the annotation used has not been constituted anywhere else yet, the creation of a testdata set had to be done manually. There are not many requirement documents openly available to the user, therefore a selection of different components, freely available via the internet, has been taken and their features wrapped up in a single document. This document has then been annotated with the help of the SemRE prototype. The evaluation takes place using an example requirement document which will be compared against the set of different components. This requirement document describes a software which is highly similar to the SemRE prototype. The set of eventually matching components also contains tools like FMOD (a sound engine) or the Graphical Editing Framework (GEF, a tool for creating graphical visualizations of models) which should therefore not get a high similarity with the requirements given. Table 5.4 lists all components which have been annotated for the purpose of this evaluation. The first column lists the names of the different components, the second one shows a short description, whereas the third one indicates if this component could be of any relevance to the requirement document.

While annotating the components it became clear that due to performance issues the

<i>Component</i>	<i>Description</i>	<i>Relevant</i>
Apache Tomcat	An application server	No
Direct Sound	Sound API, developed by Microsoft	No
EMF	Eclipse Modelling Framework, helps creating meta-models	No
FMOD	A Sound Engine	No
GEF	Graphical Editing Framework, helps visualizing any kind of model	No
GMF	Combination of GEF and EMF	No
IIS	An application server	No
Java WordNet Similarity	Library, consisting of different Java based similarity comparison algorithms	Yes
Jena	Reasoner	Yes
KAON2	Reasoner	Yes
Microsoft SQL	Database management system	Yes
MySQL	Database management system	Yes
SecondString	Library for syntactical comparison of strings	Yes
WordNet	Digital thesaurus	Yes
WordNet Similarity	Library, consisting of different Perl based similarity comparison algorithms	Yes

Table 5.4: Annotated components

amount of words in the description had to be limited. The comparison of the sample requirement document and the first IIS description, consisting of 180 words both, took more than 2.5 hours on a Core 2 Duo 3.0 GHz system with 4 GB RAM. Considering the number of total component comparisons, the overall similarity matching process would take nearly 1.5 days.

Figure 5.10 shows the overall results, gathered with the standard parameters from the last section. One thing that immediately catches the reader's eyes, is the overall number of similarities found. Most components do not seem to have a similarity with the requirement document. This is correct for the seven components, which are known to be of no similarity to the requirement document (look at table 5.4). Of those components which should show some similarities with the example document, only three have been detected. In total, 10/15 have been "identified" correctly. In more detail, three of eight similar components have been correctly detected, seven of seven dissimilar components have not shown any similarities (also correct) and five of eight similar components have not been identified as similar artefacts. The question which arises from these results is why the other similar components have not been identified. There are many possibilities:

1. The descriptions are too dissimilar (or, the other way round, the requirement document is not specific enough). WordNet just has a limited amount of words, which might not detect all semantically close relations.

2. The parameter settings are wrong. Especially the requirement similarity threshold parameter might be set too high and leave out many similarities.
3. The conceptual idea behind calculating the structural and finally the requirement / artefact similarity could be a problem. Due to their heavily ratio-based calculation method, many similarities might not be noticed. Possible enhancements are shown in 6.2.

Point one and three cannot be (dis-)proved without a lot of effort. Although there is a version 3.0 of WordNet, it can not be used under Windows yet. The third possible cause adverts to a flaw in the overall concept which also can not be fixed easily. Only the second point can be rechecked by initiating a new matching process. This time, the requirement similarity threshold has been lowered to 0.25 to gather more results. These can be seen in figure 5.11. The similarity values for all three also previously found components have been increased, but although new similarities with three other components have been found. However the better matching components can be selected more easily, based on their higher satisfaction and / or similarity values. Still the standard parameter set found in section 5.2 seems to yield better results, because it does not detect any similarities regarding the dissimilar component pairs.

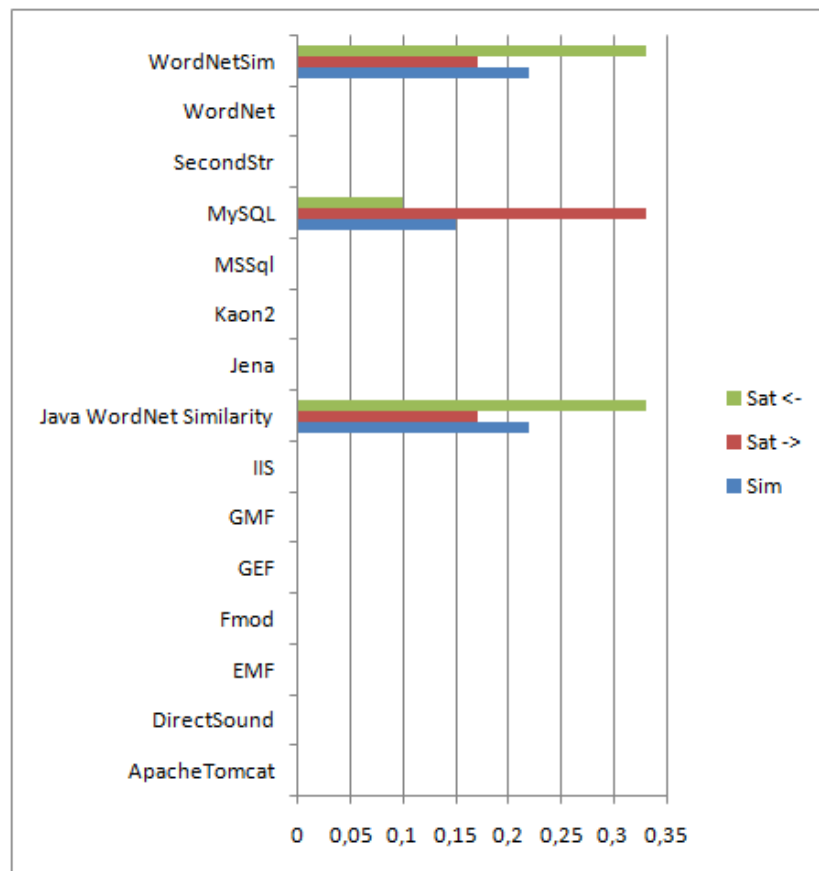


Figure 5.10: Similarity between selected components and the requirement document

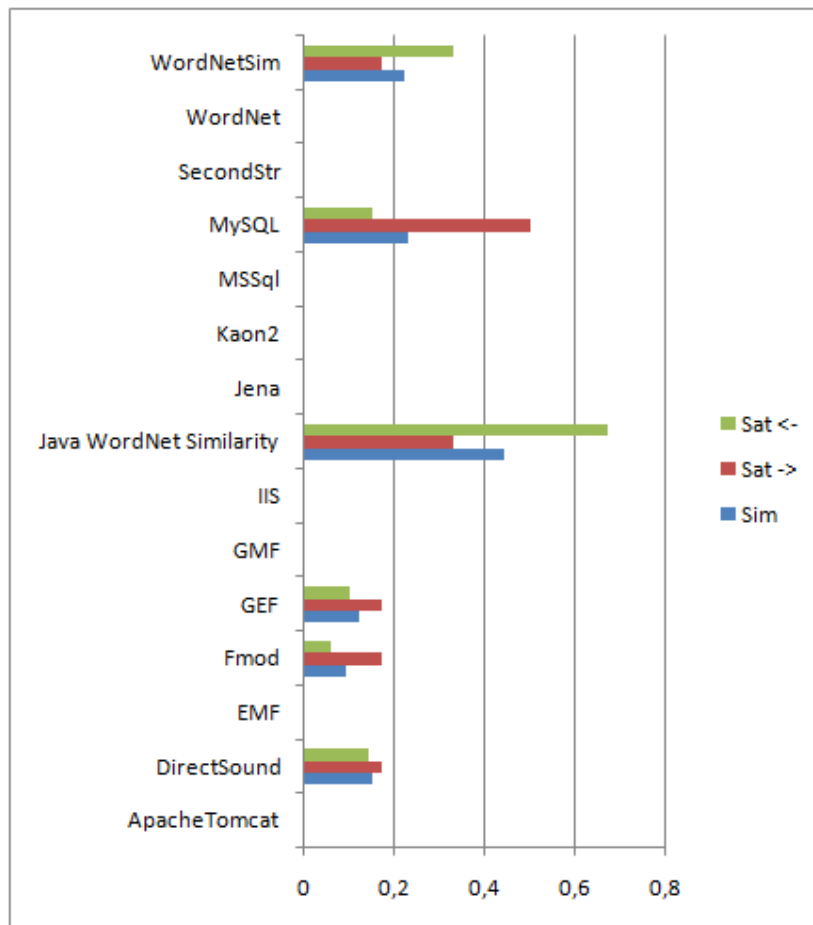


Figure 5.11: Similarity between selected components, requirement similarity threshold set at 0.25

Chapter 6

Summary

This final chapter first shows some related work, which has been done in other international projects and focusses on parts similar to this thesis. Next, some of the potential places for enhancements are represented before the whole work will be summarized.

6.1 Related work

This section gives a small overview of work which is related to the topic of this work. Namely, there are two different prior projects which will be described in more detail. In the end, the differences between both approaches will be shown.

6.1.1 Semi-automatic Semantic Annotation

To gain profit of semantic technologies a very important key to this goal is the annotation of already existing information. Erdmann et al. (see [7]) describe their experiences with the semantic annotation of content. Their primary goal was to annotate webpages with semantic tags. This was first done in a manual way which has proven to be very susceptible to faults (of a syntactic and semantic nature) and has not been well accepted by the users. The lessons they learned were that an interactive, graphical and easy to learn tool for the support of the user is a must-have. Further, they integrated an information extraction tool to further relieve the user of his task.

All of these experiences were goals which have been kept in mind during the design of the concept for this thesis.

6.1.2 CARE Approach

The COTS-aware requirements engineering approach by Chung and Cooper (see [2]) tries to solve the problem of newly created software which increases in size and therefore also in complexity, by introducing a goal- and agent-oriented software engineering

process which is centered around components. The core part of this work does not lie within the search for components, but in a complete, component oriented methodology where the requirements engineer is supported by the use of tools.

To annotate components the user needs to model the goals and agents of a component on his own in a special tool using UML and O-Telos for the creation of the knowledge base. The detailed description of a goal is done in natural language text, belonging to the corresponding goal. Next, every agent in the model is associated with a goal. The agent now tries to solve this goal on his own, i.e. he tries to select components from a repository which could match his goal. The matching is done via a simple key-word based mechanism. The user himself then has to decide if the COTS component chosen may match the requirements or not.

The main difference between the CARE-approach and this thesis is that here one focus lies on the effort to not burden the user with a new notation and / or with a lot of additional effort. Only if the user can still focus on his daily work and the system itself does everything else the user will gain real profit through the use of the system, which will in turn broaden the distribution of the software.

Additionally, this thesis does not care about the methodology as a whole, but simply focuses on the problem of finding reusable components as fast and easy as possible.

Next thing is that this thesis tries to work with state-of-the-art technologies which have evolved during the last years in the Semantic Web area. Especially the use of UML for the creation of an ontology can be problematic because UML does not provide the logical expressiveness of OWL. Also, several reasoners with different strength and weaknesses have been developed, supporting OWL.

One last point is that the CARE approach heavily focuses on COTS components and their integration, whereas the scenarios for SemRE show that it could also be used to support the structured work and search in already existing projects by finding all kind of artefact (see 2.1.5 for the definition of an artefact). SemRE can therefore be seen as a tool, which could be integrated into any software development methodology and is not bound to a very special development approach.

6.1.3 Other related work

Ehrig et al. ([6]) proposed an approach of measuring similarity between semantic business process models. Their approach also incorporates the use of a syntactic, linguistic and structural match. Their syntactic and linguistic similarity comparisons use a similar approach as the one described in this thesis except for the structural matching. It is done using a context related approach. This incorporates looking at all contextual related words and computing an overall similarity based on this context.

6.2 Outlook

During the development of the concept and the implementation of the prototype, many possible and necessary enhancements have been identified which could not have been implemented into the final prototype due to time reasons. Below several proposals are made towards future enhancements.

- One of the most urgent points is the development and integration of better NLP components, especially a better automatic coreference resolution (i.e. a better and more accurate detection of coreferences). If the results which have been shown at CoNLL 2007 can be incorporated into working software tools, the overall usability of SemRE as well as the user's acceptance could be raised.
- Another point is the use of more performance oriented reasoners. KAON2 has been introduced as a reasoner which has a high performance on large ABoxes. Due to its more difficult rule support and the lack of documentation, it has not been used for the prototype.
- The similarity comparison of the words is done using two different algorithms. However, especially the syntactic comparison is limited in its detection rate. Other algorithms like the metric, developed by A. Monge and C.Elkan ([26]), can also identify similarities between strings where one is the abbreviation of the other and could therefore be added to a whole set of similarity measure components.
- Distant versions of SemRE could also be used to automatically find contradictions and inconsistencies in the requirement document which could prevent projects from struggling in future development phases. Research on knowledge based contradiction finding has already been done at, e.g., the Xerox Parc ([31]).
- Another idea would be to enhance the complete SemRE tool with a knowledge-based traceability. By identifying a similarity between a class documentation and a requirement document, relations between both could automatically be inferred and the whole project be checked for satisfaction of the requirements.
- As already mentioned in section 3.2.3, additional inbuilt could be developed which compare NF-requirements on a specific level, e.g., if $O(n)$ lies within $O(n^2)$. A syntactic matching will show a high similarity between both terms, but it cannot identify the semantic closeness between both terms.
- Type errors are currently only paid attention to by finding words with a high syntactic similarity. Still, these words will fail if checked with the WordNet similarity measure. Therefore it should be possible to gather a set of probably matching words from WordNet in spite of the type error which can then be checked for a semantic relatedness.

- The whole process of matching ontologies for similarity is very time-consuming, because it currently computes the similarity for every ontology tuple all over again. One idea for solving this is to introduce the concept of reference ontologies (R). The reference ontology would hold a huge set of word categories. Every word of one requirement document ontology (A) is now taken and matched for similarity with one word category of the reference ontology. This way the basic similarity matching process would only have to be done once for a new ontology (currently, for two ontologies A and B) the similarity is computed from $A \rightarrow B$ and from $B \rightarrow A$.)
- The overall similarity comparison algorithm is still a very simple bottom-up approach which only considers the number of matching similarity edges and not the similarity values themselves between different words. This must be further enhanced to give more detailed results. Further, the overall structural matching process should probably be changed / compared to an approach similar to the one described in [6].
- Probably, a weighting of different word categories could enhance the results, i.e. the matching of two names should be given a higher priority than the matching of two adjectives.
- The used WordNet version 2.1 is not the most current one, yet it is the only one to work under Windows. Future versions of WordNet might contain a more detailed and matured database.

6.3 Conclusion

This thesis started by presenting an overall concept of what semantic software engineering is and what a potential IDE for SeSE (SemIDE) could look like. The scenarios of a tool for comparing different textual descriptions of components have been shown. Also its integration into SemIDE has been presented. Based on the scenarios, a concept for calculating the similarity between different artefacts has been developed which leads to the creation of a prototype. Its design as well as an evaluation of components needed has been shown in the following chapter. Because of the unknown parameter values the evaluation started with an empirical and iterative approach for finding possible parameter values. Based on these, a bigger set of artefact descriptions has been matched against a sample requirement document.

The evaluation showed that the overall concept has some potential and already obtains some promising similarity results. Still, it leaves a lot of room for enhancements. One problem is that, due to the complexity of the constructed ontologies and the comparison mechanism, the overall performance is a very poor one ($O(n^2)$). This

leads to a very long evaluation process which additionally depends on the selected parameter values. Therefore, the selected examples were relatively small regarding the number of used words. The overall performance of the system could still be enhanced by using some of the methods proposed. Especially the use of a reference ontology and / or another reasoning system could help to improve the amount of time needed to finish the matching process.

Also the detection of similar components has some flaws which results in many similar components not being detected. On the good side, dissimilar components have not been detected with the standard parameter set. This leaves a lot of room for adjustments, which have been presented in [6.2](#).

The operation on large requirement documents and components can not be proposed yet. The estimated time on normal computers or - the other way round - the financial effort to buy super-computers, still prohibits the use on larger text corpora. However, the concept in its current status has its areas: The use of only small textual descriptions still allows the operation on smaller artefacts like method or class descriptions. Larger projects consist of hundreds and even thousands of classes - finding a needed artefact in such a huge set is still a difficult task and could be facilitated, using SemRE.

List of Figures

1.1	A simple text chain	5
2.1	Modelling in SemIDE	13
2.2	Possible architecture of SemIDE	14
3.1	Integration in SemIDE	23
3.2	Projective coreferences	27
3.3	Non-projective coreferences	27
3.4	TBox structure of the basic ontology	29
3.5	A WordNet fragment	42
4.1	Overview of the prototype architecture	53
4.2	Overview of the inbuilt package	55
4.3	Overview of the xml package	56
4.4	Overview of the functional design	59
4.5	Sequence 1 - Processing of a newly loaded text file	60
4.6	Sequence 2 - Exemplary sequence of text annotation	60
4.7	Sequence 3 - Failed manipulation of text source	61
4.8	Sequence 4 - Starting of the similarity matching process	61
4.9	Overview of the NLP functionality design	62
4.10	The main window of the prototype	66
4.11	The similarity window	67
5.1	Syntactic word matching	73
5.2	Semantic word matching	74
5.3	Performance comparison of similarity threshold parameters	74
5.4	Comparison of max. path distance parameter	76
5.5	Comparison of max. path distance ratio parameter	77
5.6	Comparison of max. path distance ratio inverse parameter	77
5.7	Comparison of max. path distance ratio combined parameter	78
5.8	Overview of all found requirement pairs and their similarity	79
5.9	Similarity between basic similarity set components	80

5.10 Similarity between selected components and the requirement document	83
5.11 Similarity between selected components, requirement similarity threshold set at 0.25	84

Bibliography

- [1] Bernhard Bauer and Stephan Roser. Semantic-enabled software engineering and development. 2006.
- [2] L. Chung and K. Cooper. Towards a model-based cots-aware requirements engineering process. *MBRE*, 1:53–60, 2001.
- [3] W.W. Cohen, P. Ravikumar, and S.E. Fienberg. A comparison of string distance metrics for name-matching tasks. *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, 2003.
- [4] CoNLL. Conll 2007, 2007.
- [5] James O. Coplien. A pattern definition. 2007.
- [6] M. Ehrig, A. Koschmider, and A. Oberweis. Measuring similarity between semantic business process models. *Proceedings of the fourth Asia-Pacific conference on Comceptual modelling-Volume 67*, pages 71–80, 2007.
- [7] M. Erdmann, A. Maedche, H.P. Schnurr, and S. Staab. From manual to semi-automatic semantic annotation: About ontology-based text annotation tools. *P. Buitelaar & K. Hasida (eds). Proceedings of the COLING 2000 Workshop on Semantic Annotation and Intelligent Content*, 2000.
- [8] C. Fellbaum. *Wordnet: an electronic lexical database*. Mit Pr, 1998.
- [9] Martin Fowler. *Writing software patterns*. 2006.
- [10] Nathan A. Gilbert. *Introduction to coreference resolution*. 2007.
- [11] T.R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5/6):907–928, 1995.
- [12] H.P. Gumm and M. Sommer. *Einführung in die Informatik*. Oldenbourg, 2002.

- [13] T. Hagerup. Informatik iii, 2004.
- [14] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, 1996.
- [15] J.J. Jiang and D.W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. pages 19–33. Taiwan, 1997.
- [16] H. Kaiya and M. Saeki. Ontology based requirements analysis: lightweight semantic processing approach. *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*, pages 223–230, 2005.
- [17] T. Kudo and Y. Matsumoto. Japanese dependency analysis using cascaded chunking. *International Conference On Computational Linguistics*, pages 1–7, 2002.
- [18] M. Kuhlmann and J. Nivre. Mildly non-projective dependency structures. *22nd International Conference on Computational Linguistics and 43rd Annual Meeting of the Association for Computational Linguistics (COLING-ACL), Companion Volume, Sydney, Australia*, 2006.
- [19] Brian T. Kurotsuchi. Design pattern tutorial. 1996.
- [20] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: an initial appraisal. *CoopIS, DOA*, 2002.
- [21] D. Lin. An information-theoretic definition of similarity. *Proceedings of the 15th International Conference on Machine Learning*, pages 296–304, 1998.
- [22] R. Lo, B. He, and I. Ounis. Automatically building a stopword list for an information retrieval system. *Proceedings of the 5th Dutch-belgian Information Retrieval Workshop, Utrecht, the Netherlands*, 2005.
- [23] M.P. Marcus, B. Santorini, and M.A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1994.
- [24] R. McDonald, K. Lerman, and F. Pereira. Multilingual dependency analysis with a two-stage discriminative parser. *Proceedings of CoNLL-2006*, 2006.
- [25] R. Mitkov. *Anaphora Resolution*, volume 29. MIT Press.
- [26] A. Monge and C. Elkan. The field matching problem: Algorithms and applications. 1996.

- [27] J. Nivre and J. Nilsson. Pseudo-projective dependency parsing. *Ann Arbor*, 100, 2005.
- [28] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, 2000.
- [29] OMG. Model driven architecture.
- [30] OMG. Ontology definition metamodel. 2005.
- [31] Xerox Parc. Parc natural language processing. 2007.
- [32] S. Roser, F. Lautenbacher, and W. Fischer. Semse vision. 2007.
- [33] Ian Sommerville. *Software Engineering*. Pearson Studium, 2001.
- [34] Phil Tetlow, Jeff Z. Pan, Daniel Oberle, Evan Wallace, Michael Uschold, and Elisa Kendall. Ontology driven architectures and potential uses of the semantic web in systems and software engineering. 2006.
- [35] W3C. Owl, 2004.
- [36] W.E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. *Proceedings of the Section on Survey Research Methods, American Statistical Association*, pages 354–359, 1990.
- [37] W.E. Winkler. The state of record linkage and current research problems. *RR99/03, US Bureau of the Census*, 1999.
- [38] H. Yamada and Y. Matsumoto. Statistical dependency analysis with support vector machines. *Proc. IWPT*, pages 195–206, 2003.
- [39] P. ZAVE. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4), 1997.
- [40] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev. An ontology-based approach for traceability recovery. *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006), Genoa, October*, 1:36–43.

Chapter 7

Appendix

7.1 Usage instruction for the GUI

As soon as the user has starts the software he can load a file via clicking on "File" → "Open". Depending on if the file has not been annotated already, SemRE starts detecting coreferences and possible NF-requirements. After the process has finished, the user can start annotating the text. Every possible action is done by first marking a certain text fragment. After that, he can choose from several different options on the left side. Either he marks the selected text as a new (NF-) requirement or a coreference. This is done by simply pressing the corresponding buttons. If he wants to delete a certain tag the user just clicks on the wrongly annotated text and presses the "Delete" button in the lower left corner of the window. Whenever he wishes to he can save the currently loaded text. To do so, he can either press Ctrl + S or click on "File" → "Save". This will result in the last known file being overwritten with the current text. If the user wishes to store the current data in another directory, he must click on "File" → "Save as...".

The user is also given the ability to directly edit the text. Therefore he must switch the view from "Normal Text Rendering" to "Source Text Rendering". Further, the tags can be edited in there. However, this is not recommended!

For starting the similarity search dialogue he must click on "Similarity" → "Find suitable components...". This opens a new window, in which the user can specify different settings. The first parameter is the path to the ontology directory. Next, the different parameters described in 5.1.1 can be changed. After that he simply must click on "Start Search" for the matching process to begin. It may take some time for the first results to appear in the topmost table. As soon as the first result is displayed it can be exported by just pressing the "Export" button. This creates a new ASCII-based file which contains all received information. Further he can parse these kinds of information by selecting the corresponding entry from the upper table. This will result

in both textual descriptions being displayed in the textareas as well as all matching requirements appearing in the table in the middle. To get all similar words of two matching requirements the corresponding line must be selected in the middle.

7.2 Scenarios

This section lists potential scenarios which are typical for today's as well as future's requirements engineering.

7.2.1 Nowadays - Scenarios

This section describes scenarios (1 to 5), which are possible nowadays. Because of the still existing separation of requirements engineering and component search, these two actions are separated into different subsections.

ID	1
Role	End user
Condition	<ul style="list-style-type: none"> - Needs a new software - Doesn't have knowledge of software engineering - Uses some kind of word processor software
Action	Simply writes down requirements in unstructured form
Result	- Document with at least partially ambiguous semantics and incomplete / missing requirement descriptions as well as wrong technical terms.

ID	2
Role	End user
Condition	<ul style="list-style-type: none"> - Needs a new software - Has some knowledge of software engineering - Uses some kind of word processor software
Action	Simply writes down requirements
Result	- Document with some structure and more focus on what is relevant. Problems may be identifiable. Perhaps clearer semantics

ID	3
Role	Requirement engineer
Condition	- Knows the problem space - Uses word processor
Action	Writes down specific requirements, which exactly describe all needed features of the new software
Result	- Document with structured and mostly semantically unambiguous requirements

ID	4
Role	Requirement engineer
Condition	- Knows the problem space - Uses special requirements engineering tool
Action	Writes down specific requirements, which exactly describe all needed features of the new software
Result	- Document with structured and mostly semantically unambiguous requirements. Requirement will probably be reviewed by other team members and as such even become clearer.

ID	5
Role	Software Engineer / Requirement engineer
Condition	- Knows the problem space - Small changes should be made to an existing software project
Action	Probably none. Small changes may be seen as irrelevant and just be told on a vocal basis.
Result	- Requirement document becomes incomplete and inconsistent.

7.2.2 Nowadays - Component search

This section describes possible scenarios (6 to 13), which can happen, regarding the component search problem in the beginning of a new project.

ID	6
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming did not start yet - Semantically ambiguous requirements
Action	Does not care for probably existing components (time / motivation / political reasons) and simply starts developing
Result	<ul style="list-style-type: none"> - Pure / less quality than when using existing components - More time needed for development of software

ID	7
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming did not start yet - Semantically ambiguous requirements
Action	Selects only components which are already well known to him
Result	<ul style="list-style-type: none"> - Selected components might not exactly solve the problem - Probably better components have been developed in the meantime - Could not find certain components which would satisfy certain requirements due to time / motivation etc.

7.2.3 Intended scenarios

This section describes many possible scenarios (14 to 18), which can be imagined, using a SemRE tool.

ID	8
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming has started - Programmer needs a software routine - Semantically unambiguous requirements
Action	Searches the framework library and / or uses an internet search engine to find a suitable method, but cannot find one (in a certain amount of time). Starts writing the method on its own
Result	<ul style="list-style-type: none"> - Self written method with possible bugs due to inefficient testing - Time lost with something that had no results

ID	9
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming has not started yet - Semantically unambiguous requirements
Action	Makes a quick search in a search engine for key words ("Database", "Persistence layer" etc.)
Result	<ul style="list-style-type: none"> - Finds basic components in a justifiable amount of time, which can be incorporated into the software design - Might not have identified all possibly available components - Components might not exactly meet his needs

ID	10
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming has already started - Semantically unambiguous requirements
Action	By accident finds a suitable component and starts rewriting the software
Result	<ul style="list-style-type: none"> - Probably a lot of time needed to rewrite the existing source code - New bugs evolve - Embedding of the new component into the software could deteriorate the existing software design

ID	11
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming has not started - Semantically unambiguous requirements - Wants to search for all possibly matching components
Action	Takes time to intensively search for components that cover a huge set of his requirements. Incorporates the results into the design phase
Result	<ul style="list-style-type: none"> - Probably needs a lot of time for an intensive search - Gains time by not having to write components on his own - Leads to a better design of the software

ID	12
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Programming has not started - Semantically unambiguous requirements - Needs a software routine
Action	Searches the frameworks library and/or uses Google for a suitable method and finds it
Result	<ul style="list-style-type: none"> - Probably needs a lot of time for an intensive search - If the method was found via Google (although there might be a better solution in the framework), the solution might have flaws and might also not have been tested efficiently - Perhaps he might get legal problems (copyright / trademark etc.)

ID	13
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Requirements document exists - Semantically ambiguous requirements - Needs a software component
Action	Searches for components that match his understanding of the requirements and finds some.
Result	<ul style="list-style-type: none"> - "His understanding" might not have been the understanding of the person who has intentionally written the requirements - Could lead to a huge problem (depending on when this issue will be discovered (the earlier the better)).

ID	14
Role	End user / requirement engineer
Condition	<ul style="list-style-type: none"> - Using SemRE - tool - Wants to describe the features for a new software product
Action	User writes and annotates its requirements in a special editor which puts all the information directly into an ontology
Result	<ul style="list-style-type: none"> - More time needed for annotating requirements - SemRE tool automatically searches for components which satisfy some of the requirements. Those components then can be directly incorporated into the design-phase

ID	15
Role	Software engineer
Condition	<ul style="list-style-type: none"> - Using SemRE - tool - Wants to find a method which satisfies certain requirements
Action	User specifies the features in analogy to specifying the requirements for the requirement document. The SemRE tool automatically parses the information, adds them to the ontology and searches for matches.
Result	<ul style="list-style-type: none"> - More time needed for annotating requirements - SemRE tool automatically searches for a corresponding method in the whole project, which satisfies the user's requirements.

ID	16
Role	End user / RE
Condition	<ul style="list-style-type: none"> - Using SemRE - tool - The requirement document already exists - Wants to get as much information as possible from the document
Action	User imports the completed requirement document into the SemRE tool, which automatically extracts all relevant information from it and uses them to search for suitable components.
Result	<ul style="list-style-type: none"> - As fast as normal user requirements engineering - SemRE Tools automatically search for components which satisfy some of the requirements. These components can then be directly incorporated into the design-phase.

ID	17
Role	End user / RE
Condition	- Using SemRE - tool - Wants do describe his needs for a software
Action	User annotates its requirements in the SemRE tools, which automatically searches its repository for possibly matching software components.
Result	- User gets a set of existing and matching software products / components, which exactly fit his needs. - More convenient for the user, because the search is more efficient and faster than using any kind of search machine.

ID	18
Role	Software engineer
Condition	- Using SemRE - tool - Wants do write a method which satisfies certain requirements
Action	User starts by writing the documentation of the method. The tool parses this information and searches for a possible match.
Result	- The user automatically gets the results needed. - Avoidance of redundancy