Signature Matching by Concept Contraction and Concept Abduction

by

Kai Müller

Draft

Submitted to the Technical University of Hamburg-Haburg in partial fullfillment of the requirements for the degree "Master of Science in Information and Media Technologies"

supervised by

Prof. Dr. Ralf Möller (STS)

Prof. Dr. Volker Turau (TI6)

March 2007

Abstract

The advent of the information (or knowledge) society has brought new problems and challenges. There is a kind of information overflow as information gets accessible by more and more people. Since the invention of the internet it is as easy as never before to make one's knowledge available to other people.

Knowledge representation systems take over the responsibility to store these huge amounts of knowledge and present ways to access and retrieve them. Those systems work, but only under certain conditions. The biggest problem is actually not storing information but accessing it in an effective and efficient way. Thus without appropriate ways to search and retrieve information the usefulness of these systems is questionable.

One way to address this problem is to look at how content is classified when it is stored in the knowledge representation system. This master thesis presents signature matching as and efficient and effective way to classify medial content. The aim is to look at what kind of matches are usually required by users of these systems and then develop a method to foster this approach already during the classification process.

This said during searching activities it is often required from search algorithms, that not only perfect but relaxed (meaning not 100% fitting) matches are returned. This master thesis takes a description logic system as the basic content classification system and extends it with means to achieve this goal. The advanced inferencing services "Concept Contraction" and "Concept Abduction" are employed to find relaxed matches and also give a ranking of the returned results.

Declaration

I hereby declare to have written this master thesis entirely by myself and used no other sources than those listed in the resources part at the end of this thesis. Furthermore, I affirm that this thesis has never been published in Germany or in any other countries.

Hamburg, 23rd March 2007

Kai Müller

Acknowledgements

I would like to thank Professor Ralf Moeller of the STS department for supervising the thesis. Thanks also go to Professor Volker Turau of the TI6 department for being the co-corrector.

I would also like to thank Sebastian Boßung of the STS department for the endless discussions about the signature model and general topics, LaTeX especially.

Contents

List of Figures

Chapter 1. Introduction

1.1. Motivation

In our society knowledge has become one of the most important resources. A whole "knowledge economy" has developed around the fact that knowledge has become tradable, because a) the storage facilities have improved and b) the distribution has become very easy and cheap through the advent of the Internet. But with a wealth of knowledge and new ways to handle it also new problems arise. The amount of knowledge is overwhelming and it can be hard to utilize it efficiently. Search and retrieval of knowledge are especially sensitive topics, because these directly contribute to the value of the newly available knowledge.

Search and retrieval functionality can be approached upon from two sides: Top-down, where one builds upon existing databases (or other means of storing knowledge) and tries to find efficient ways to access the available data by developing efficient algorithms. Though it is quite easy to experiment in this area and results come fast, it does not go to the roots of the problem. The alternative is a bottom-up approach where one examines the way knowledge is stored and organized in the first place. The structuring of knowledge is an important criteria for efficient access. In this case structure of knowledge means powerful mechanisms to classify knowledge. Only through a thorough classification, possibly in a (semi-)automated way, the full power of storable knowledge can be exploited.

One imminent area of concern is the publication of content via the internet. People are now able to become their own content producers and these possibilities are used by many. "*Content*" in this context means multimedial content, like pictures, texts, videos, music, etc. or more specifically it means digital representations of all these types of content. It is as easy as never before to create a digital representation of the medial content one has produced and to publish it in the internet. This process is of only limited value if no effective and efficient means to search and retrieve are available. The interesting point here is that storing content and storing knowledge are interrelated. Any means of retrieving medial content requires the storage of knowledge about the content beforehand.

The goal of this classification is to use information gained in previous

instances of classification (i.e. context knowledge) to a) classify correctly and b) classify efficiently. One can find this approach also in the principle of reuse of software components in the object orientation paradigm of computer science. The basic idea is to reuse components so that software becomes cheaper to develop, is of higher quality and is more flexible to use. This is only made possible because of means to find the right component for the right task.

One solution to this categorization problem is signature matching. In the area of software development, signature matching compares signatures (the structure) of software modules (or the methods of modules) to find the most appropriate counterpart. This means search is not done as usual by only comparing names, but by taking context knowledge into account.

If one wants to utilize signature matching for the classification of arbitrary medial content, the first problem is the translation of content into a signature. This is where description logics excel as means to model information domains and store both syntactic and semantic information. Description logics are well-known and wide-spread systems to model and store knowledge. They also allow (somewhat limited) reasoning over the stored explicit knowledge to gain additional implicit knowledge.

Examples where these technologies are already employed are E-marketplaces. Here signature matching is used to find negotiation spaces for suppliers and consumers. Signature matching helps in providing means to find a common ground even if the things searched for and the things provided do not match exactly. Other example are applications which work with user profiles like pages offering job exchanges or dating services. They use the information gathered about a user to find appropriate matches. User profiles then work as a kind of signature and signature matching algorithms are employed to find the best matches.

All these approaches work well in their own domain, but they have a common limitation. Standard signature matching works well where exact matches are the goal, but when dealing with arbitrary types of content the limited number of results is unsatisfiable. With the help of description logics and advanced matching algorithms for relaxed or fuzzy matching this problem can be solved.

This master thesis takes the approach of signature matching a step further and extends it with advanced functionality for relaxed matching.

Through this it is possible to utilize the power of signature matching in the fuzzy world of knowledge management.

1.2. Structure Of This Thesis

This thesis presents the results of developing a signature matching algorithm that employs concept contraction and concept abduction for relaxed matching. It can be roughly divided into three parts.

The first part explains the problem at hand. In chapter 2 the problem is specified and the objectives of the thesis are explained. It also gives an introduction into the problem domain used for the implementation of a proof of concept application. Chapter 3 relates the thesis to other work done in this area and points out similar approaches to the problem. As the thesis should be complete in itself chapter 4 gives an introduction into all relevant topics and concepts used throughout the rest of the thesis.

The second part is the core part. Chapter 5 explains the solution to the problem. The solution consists of two main ideas: A signature model for modeling content signatures and a matching algorithm that employs concept contraction and concept abduction to rank possible results. To prove the viability of the algorithm a prototype application was developed. Chapters 6 gives detailed information about the decisions made during design and implementation of the sample application. Chapter 7 contains an evaluation of the developed prototype.

The master thesis is concluded with chapter 8, which shows the conclusions drawn from the whole work. It contains observations made throughout the thesis and gives an outlook on future research directions and topics.

Chapter 2. The Problem

Summary. *This chapter elaborates some of the main problems, that arise when one tries to match arbitrary types of medial-content. It shows why the problems can be related to matching signatures and why signature matching can be applied to more than just matching function signatures of software components. It explains the objectives of the thesis as well as the application domain used for the reference implementation of the algorithm.*

2.1. Problem Description

As mentioned before search and retrieval are very important topics when talking about the efficient use of stored knowledge. Among systems that are able to store knowledge, *description logic* (DL) systems are very popular. They are very flexible, customizable and allow to discover information that is not visible in the first place, with the help of so called inferencing services. After one such system has stored all the important knowledge, the question arises of how to retrieve exactly that information that one needs right now. This is where signature matching comes into play.

One can differentiate two types of signature matching: exact matching and relaxed matching. The first can be very well executed by DL systems, but it only delivers 100% perfect matches (meaning that the matches equal every aspect specified in the search). This is how most matching procedures operate until now. The second one offers ways to get not so perfect matches as possible matching candidates. This is necessary because when managing knowledge one has to take two important things into account: 1. Users are humans who make mistakes and 2. a lot of knowledge is dependent on contextual information (of the user or another person). Thus finding not so perfect matches is preferred to just finding exact matches.

A problem arises here, because the standard services provided by DLs (namely satisfiability and subsumption, see chapter 4) are not enough to reach this goal. They allow for basic relaxation (for example subsumption allows to find results which are not exact, but which are more general than the searched result), but they fail when the relaxed search needs to be more

complex. What if a result contains everything searched for except one little part? Or what if it fits otherwise perfectly but there exists a single contradiction between one component?

Finding more matches through relaxed matching does not solve the problem alone. What is also needed are means to evaluate the results. This means that the results should be sorted in a way which expresses the degree to which they fit. This requires also that the measurements are transparent to the user, so that he can make an optimal decision.

2.2. Objectives Of The Master Thesis

This master thesis has the goal to develop and implement a signature matching algorithm for matching arbitrary types of content. The algorithm will contain two different approaches: Firstly "exact matching", which is used to identify perfectly matching counterparts and secondly "relaxed matching", which is employed to find and rank matches that do not fit perfectly. The latter part uses "*Concept Contraction*" (CC) and "*Concept Abduction*" (CA) as means to identify these relaxed matches.

The algorithm will be developed as generic as possible, to allow it to be used in as many application contexts as possible. The main objective can be divided into three sub-objectives:

Development Of A Signature Matching Algorithm

This sub-goal consists of the basic implementation of the signature matching algorithm. It includes the development of a signature model, which is then used by the algorithm. In the first instance this algorithm will only support exact matching of demand and supply signatures, and will only use the standard inferencing services of the used DL system.

Relaxed Matching By Concept Abduction And Concept Contraction

In this step the signature matching algorithm is extended with relaxed matching functionality. Based on CC and CA partial matches are identified. This means that this step includes the development of CC and CA as extensions for the DL system.

Ranking Of Results With Transparent Penalty Functions

To increase the use of the relaxed result sets, a ranking function is implemented. It is based on a penalty function, which assigns penalties for the transformations made during CA and CC. The ranked result set shows how close a relaxed match is to the original query. In this step several penalty functions will be implemented and evaluated to see how they influence the results.

2.3. The Example Domain

To be able to make a real-world verification and evaluation of the developed algorithm a concrete example domain is used. This section introduces the domain and explains the signature problem inherent to its classification schema.

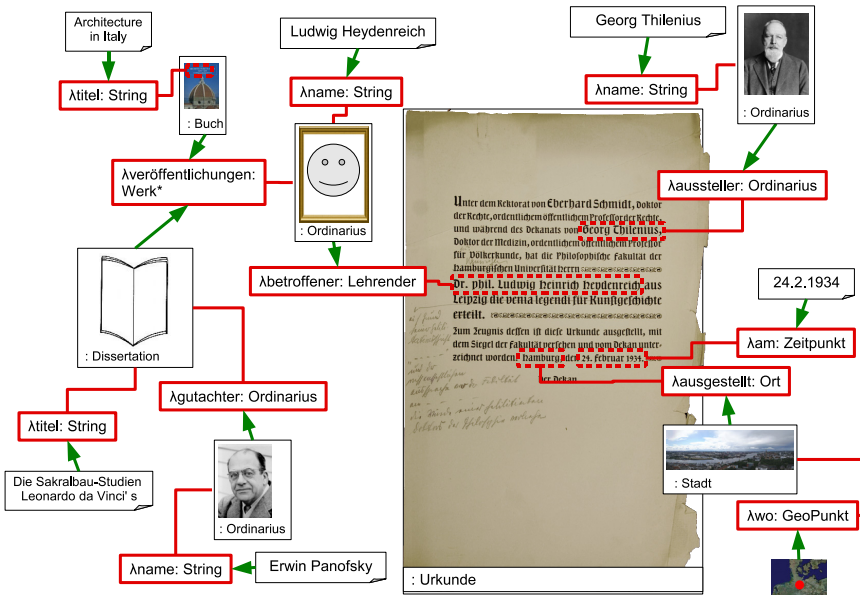


Figure 2.1: Venia-Legendi example classification (from [Boßung, 2007])

The Technical University of Hamburg Harburg is developing together with the art art history departments of the universities of Berlin, Bonn, Munich and Hamburg a networking infrastructure for data about historical art. It is based on the "*Warburg Electronic Library*" (WEL, see [wel]). The users of the WEL have access to multimedia documents and are able to configure their use individually for their needs. Example projects include

a digital library for political iconography and the "*Geschichte der Kunstgeschichte im Nationalsozialismus*" (GKNS, see [gkn]). GKNS is dealing with the creation of a thematic network, about the history of art in the national-socialist era. Thus the type of content that the GKNS busies itself with, are reproductions of all kinds of artworks.

Content is described by means of an "Asset Expression Model". Asset expressions are based on the λ-calculus and allow for a typed classification of components of content. Figure 2.1 shows an example classification of a government document. The problem with classification of historical content stems from the necessity of contextual knowledge. Different people possess different contextual knowledge. This also includes a lack in some specific area or contradicting knowledge.

As with software components, the real value created by the GKNS comes from the search and retrieval functionality. While the GKNS helps with features for content classification, this process is still done manually and offers no functionality for automated classification. The problem is inherent to data requiring contextual knowledge. While it may be easy and obvious to classify a portrait or a passport, complex medial content (like the one shown in the example above) have a context dependent meaning. One way to approach this problem, is to see it as a signature matching problem within the GKNS. Whenever new content is added to an already existing database, what is the best possible classification done before, that identifies the new piece of art?

Chapter 3. Related Work

Summary. *Signature matching has its origins in the field of computer science and is related to the object-oriented programming paradigm. This chapter shows alternatives for its use and relates them to the thesis. In addition, alternative approaches to the problem of content matching are highlighted and commented. Finally alternative uses for signature matching are discussed.*

3.1. Description Logics And Signature Matching

One area where matching of signatures is very important, is in applications where data about users and their behavior is collected and stored. Examples come from many different application domains: Online-stores use behavioral data about their users for personalized advertising; dating systems offer users advanced searching algorithms to find the "perfect" match to their own profile; job recruitment pages offer companies as well as people who search a job possibilities to create, maintain and match job profiles. The difficult task in all these systems is finding and matching user-profiles to a given query profile. This problem becomes worse as in all these applications a perfect match is almost impossible, because the possibilities for profiles are almost endless. The core problem is that profiles are often incomplete or incompatible with other profiles. These are common problems when matching user profiles and have to be taken into account. Thus one needs to alter the search such that the "best possible" match is searched for.

Cali, Calvanese, Colucci, Di Noia and Donini approach this problem in [Calì et al., 2004] and propose a description logic system specifically tailored to representing user profiles. It makes use of the advanced inferencing services concept contraction and concept abduction to offer the users transparent criteria for the ranking of a result set. Their basic idea is that in the presence of missing and conflicting information concept contraction and concept abduction are the solution to both – finding fuzzy matches and assigning proper rankings. Their description logic system is a basic $\mathcal{AL}$ description logic, with the extensions:

- Full existential qualification:

  This extension allows for existential restrictions on profile data and is used to express whether a user has interest in a specific topic.

- Predicate restriction:

  This allows the assignment of concrete values (called levels) to specific concepts and is used to express the level of interest a user has in a specific topic.

User profiles are then represented in the system as a conjunction of the following:

- Atomic concepts, representing atomic properties about the user, like gender.
- A conjunction of predicate restrictions representing physical characteristics, like age or height.
- A conjunction of concepts of the form hasInterest showing the least and the most interest a user has in a specific area.

An example profile of a 25 year old man, with strong interests in Japanese comics, fantasy novels and politics but absolutely no interest in soccer is shown here:

$$
\begin{aligned}
&male \sqcap =_{25} (age) \sqcap =_{1.96} (height) \sqcap \\
&\exists hasInterest.(fantasyNovels \sqcap \geq_{0.9} (level)) \sqcap \\
&\exists hasInterest.(japaneseComics \sqcap \geq_{0.9} (level)) \sqcap \\
&\exists hasInterest.(politics \sqcap \geq_{0.9} (level)) \sqcap \\
&\forall hasInterest.(\neg soccer \sqcup \leq_{0} (level))
\end{aligned}
$$

The proposed matching algorithm takes a demand profile and searches through all available supply profiles. It uses concept contraction to filter out contradicting concepts. A contradiction would for example appear if a profile which states an interest in politics with a level of “at least 0.5” is compared with the example profile above. After all contradictions are filtered out, concept abduction is used to find out which interests the supply requires but which the demand cannot deliver. This would be the case if for example a profile

which states an interest in German culture of "at least 0.5" is compared with the example profile above. Both, contradictions and missing information, are penalized through penalty functions, which calculate penalties for the changes made during the contraction and abduction phases. Based on these penalties a ranking of the profiles that were compared is created.

This approach is similar to the one taken in this thesis because the same advanced inferencing services are used to match the profiles. Through the use of a proprietary description logic though, the algorithm for matching the user profiles is not very flexible. In addition no analysis regarding the effectiveness of their chosen penalty function was done.

In [Colucci et al., 2004b] almost the same team of scientists addresses the problem of negotiation spaces in e-marketplaces. The problems of matching offers and requests in this domain are similar to the one mentioned above, but differ in one important aspect. In this area perfect matches almost never happen, because of requirements that are not negotiable for one party or the other. This means the parameters of an offer/a request have to be classified into different categories. Thus a very interesting part of the proposed description logic design is the distinction of "*strict*" (ST) and "*negotiable*" (NG) requirements. This is what opens up negotiation spaces, as it is now possible to distinguish an offer that does not match at all (ST requirements are not fulfilled) from one that matches partially (ST requirements are fulfilled, but the NG requirements are not). Through the distinction of the two types of requirements requests and offers get an inner structure. It is thus no longer necessary that the offer and the request match perfectly.

The proposed algorithm is based on description logics, which allow for an "open world assumption" (which means, that one accepts that knowledge in the world is incomplete and thus that absence of information does not necessarily mean wrong information). This has two distinctive advantages: Firstly, incomplete information is allowed and secondly, absence of information can be distinguished from negative information. The algorithm is based on a standard $\mathcal{ALN}$ description logic and it makes also use of the advanced inferencing services concept contraction and concept abduction.

A second interesting point is the rankPotential method which is used to create a ranking of the results. This method is a kind of penalty function. It assesses how "close" two concept descriptions are, and quantifies this closeness as a number. The algorithm compares two concepts (C and D)

and analyzes three differences:

1. How many concepts appear in D, that do not appear in C?
2. Are all number restrictions of D represented in C?
3. If universal role qualifications are given by D, are they mirrored by C? If not how close are the qualifications from C to the ones from D?

These considerations are similar to the ones made throughout the development process of the penalty functions proposed by this thesis. Basically the algorithm analyzes which restrictions are demanded by D that are not met by C. However the algorithm is using fixed weights for every portion of the assessment and no investigation regarding the accuracy of the penalties has been made.

In [Di Noia et al., 2003] Di Noia, Di Sciascio, Donini and Mongiello propose the use of concept abduction for several recent matchmaking problems. Their major example is human matchmaking in the form of apartment ads in newspapers that are matched against user preferences for a new flat. Other quoted examples that could benefit are software agent matchmaking, matchmaking in e-marketplaces and service discovery.

They base their work on four principles for matchmaking, which are also relevant for this thesis:

Open World Descriptions

Open world descriptions come with the use of description logics, and are based on the open world assumption. This makes it possible to distinguish absence of information from negation of information. Absent information should then be interpreted as either something that has to be refined later, or something that is of no importance.

Non-symmetric Evaluation

Non-symmetric evaluation relates to the fact that it is possible that a supply S is a perfect match for a demand D, while D is no match at all for S. This should be mirrored in the ranking D and S get respectively to each other.

Syntax Independence In Ranking

This is made possible through the use of a description logic. As concepts are related to each other it is possible to create syntax independence of ranking

results. If for example one supply specifies the months "June-July-August" it should be equally ranked to a supply that specifies "Summer", if these supplies are otherwise equal. This property is a very strong claim, which depends heavily on the semantic meaning of the defined concepts. Though implication and subsumption guarantee a certain relationship between concepts it is foremost of syntactic and not semantic character.

Monotonicity In Ranking Over Subsumption

The subsumption relationship also directly translates to the ranking of two supplies S_1 and S_2. If S_1 subsumes S_2, S_2 should get better ranking, because it is more specific and thus fits the demand better. This again is a very strong claim, which may not be true always. A counterexample is if S_2 introduces a contradiction with the specified demand.

The algorithm proposed is based solely on concept abduction. The ranking of the returned results is thus only based on the length of hypothesis created through concept abduction. This is a very shallow approach as the "types" of concepts in the hypothesis are not taken into account (which could then lead to the mentioned contradictions).

3.2. Advanced Inferencing For Signature Matching

Concept abduction and concept contraction are only two possible enhancements for description logics with regards to the "distance" measurement of concepts. Several other advanced inferencing services have been developed (see [Brandt and Turhan, 2001]) and this section introduces two alternatives which could be used to solve the problem at hand.

3.2.1 Concept Difference

Teege introduced in [Teege, 1994] the "difference" operation for description logics. The goal was simple: compare two concept descriptions and remove as much as possible of the information contained in one description from the other description. Informally this means that a new description is created, which contains all info that is part of one and not the other description. As this method can also be used to remove unnecessary information it is also called "subtraction". Teege takes three types of expressions into account for his calculations. Concepts (descriptions of objects), roles (descriptions of relationships between objects) and features (descriptions of functions between

objects). The result is a new expression, but it is important to state, that it does not extend the description logic in any way with expressions not covered by the three types mentioned above. The difference operation is based on concept conjunction and the basic inferencing service subsumption, which makes it independent of the specific type of description logic used (though some description logics require special rules). The formal definition of the difference operation is:

Definition 1 *Let $\mathcal{L}$ be a description logic. Let $\sqcap$ denote the conjunction operation in $\mathcal{L}$, let $\sqsupseteq$ denote the subsumption relation in $\mathcal{L}$ and let $\equiv$ denote semantic equivalence in $\mathcal{L}$. $max_{\sqsubseteq}$ denotes the maximal concept w.r.t. subsumption. Let A,B $\in \mathcal{L}$ be two descriptions in the logic with $A \sqsupseteq B$. Then the **difference** B – A **of** A **and** B is defined by*

$$B - A := max_{\sqsubseteq} \{C \in \mathcal{L} : A \sqcap C \equiv B\}.$$

*The set $\{C \in \mathcal{L} : A \sqcap C \equiv B\}$ is called the **difference candidates** and is denoted by $B \ominus A$.*

Every description C in the result contains enough information to yield the information in B, if added to A i.e. it contains all information of B which is missing in A. In addition every C is maximally general, meaning it does not contain any unnecessary information. Semantically this means that C contains every individual of B and no individual of $A \setminus B$. Two extreme cases exist, which are shown in figure 3.1. a) is the case if $C = B$ and b) if $C = B \sqcup (\mathcal{D} \setminus A)$, where $\mathcal{D}$ represents the domain knowledge.

Figure 3.2 shows a simple example, where B consists of the atomic concepts x,y and z and A is the single atom x. The difference is the conjunct of y and z.

Concept difference is another approach to measure the distinction or commonality between different concepts. Teege shows that it is possible to find the (semantic) maximum if the description logic used meets specific conditions.

A second approach to concept difference is presented in [Brandt et al., 2002]. The goal was to “translate” concepts from one DL to another. This is done by creating an approximation of the concept in the new DL. In contrast to Teeges approach not the semantic maximum is searched but the syntactic minimum.

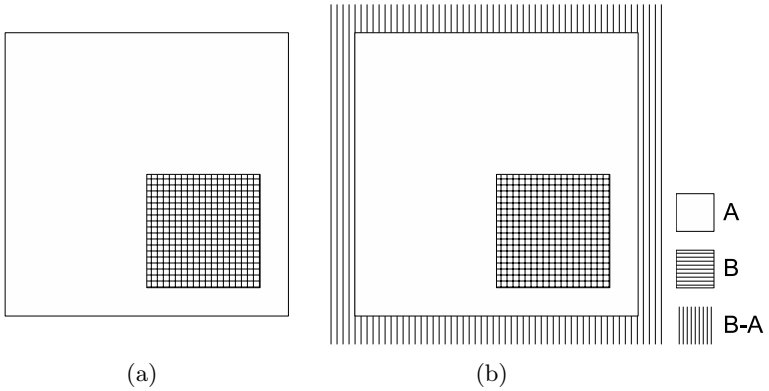


Figure 3.1: Extreme cases of the difference operation. (from [Teege, 1994])

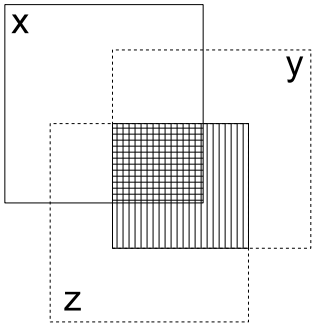


Figure 3.2: Example difference of concept atoms. (from [Teege, 1994])

Both approaches also work toward solving the problem presented in this thesis, though they have no means to actually measure the result. The new concept created describes the difference between two concepts perfectly well, but there is no way to actually qualify the results. Some method to measure the results of several queries against each other is missing.

3.2.2 Least Common Subsumer

Another inferencing service which could be employed to measure the difference of concepts is the "least common subsumer" (LCS) operation. This service finds the maximum description that all descriptions of a given collection have in common. Formally this is the most specific concept that subsumes all the descriptions of the collection. What this description looks like and whether it exists depends strongly on the description logic under

consideration. The formal definition of the LCS is:

Definition 2 *Let $\mathcal{L}$ be a DL. A concept description E of $\mathcal{L}$ is a least common subsumer (LCS) of the concept descriptions $C_1, ..., C_n$ in $\mathcal{L}$ ($(LCS_{\mathcal{L}}(C_1, ..., C_n)$ for short) iff it satisfies*

1. *$C_1 \sqsubseteq E$ for all i = 1,...,n, and*
2. *E is the least $\mathcal{L}$ concept description with this property, i.e. if E' is an $\mathcal{L}$ concept description satisfying 1., then $E \sqsubseteq E'$.*

The LCS – if it exists – is always unique and it is sufficient to define a binary LCS of two descriptions, because n-ary LCS can be defined recursively. In [Baader et al., 2005] it is shown that the LCS does not always exist. More specifically it does not exist if one of the following scenarios is true:

1. There may not exist a concept description, that satisfies 1. of the definition.
2. There my be several subsumption incomparable minimal concept descriptions satisfying 1. of the definition.
3. An infinite chain of more and more specific concepts can exist, that all satisfy 1. of the definition.

The first scenario can not happen if the top concept is part of the description logic. The second cannot occur if conjunction is available as a constructor in the description logic. The last one can happen in a TBox with cyclic concept definitions.

The LCS was topic of many researchers in the last years (refer to [Baader and Küsters, 1998], [Dean, 1999], [Baader et al., 2005], [Mantay, 1999] and [Brandt et al., 2002]), and it was shown, that the LCS not only exists for expressive description logics up to $\mathcal{ALEN}$, but that it can be effectively calculated. In [Baader and Küsters, 2000] Baader and Küsters analyze the matching potential of the LCS operation and explore the complexity if applied to expressive description logics. In [Möller et al., 1998] the LCS is successfully used to measure the difference between two concepts.

This thesis requires a description logic of at-least the complexity of an $\mathcal{ALEN}$, which means the LCS can be used to replace concept contraction

and concept abduction, but it requires a quite complex operation. In addition the LCS has also the problem of not being able to qualify the results. This is because finding the LCS does in itself not yield a measure for the difference of two concepts. It is still necessary to compare the found LCS with the existing concepts, and apply some sort of difference operation. Another solution might be to create a ranking solely on the LCS found during the algorithm, but that again would require some means to measure the difference of different LCS.

3.3. Other Applications For Signature Matching

In [Zaremski, 1996] several other applications for signature matching are proposed. Namely content browsing, content indexing and content substitution. Although only applications in the context of software libraries are discussed, it is not hard to think of similar applications for other types of contents.

One can for example imagine browsing in an art library where one can add/remove specific requirements through the creation of a signature. Users of this library could create personal profiles, containing specifications about the type of art they like. The application would then create their own exhibition based on the profile.

Another example where this is already done but with different methods (mostly manually), is the creation of catalogs in libraries. Users of a library can already use catalogs based on the title of the book, the name of the author or sometimes even the content of the book. Based on signatures and profiles users could create their very own catalog according to their specific needs.

An example where substitution could be helpfully implemented through signature matching is if an art gallery wants to make an exhibition about a specific topic. Now one piece of art is not available and the gallery has to search for a substitute. The algorithm would search through all available art catalogs and present a ranking of possible alternatives.

All these examples show that signature matching has its relevance outside of matching function signatures of software component libraries.

Chapter 4. Background

Summary. *The algorithm presented builds upon two main concepts: signature matching and description logics. In addition the example domain uses the "Asset Expression Model", which is based on the λ-Calculus. To be self-contained, this chapter gives an introduction to the used concepts and provides the reader with background information for the rest of the thesis.*

4.1. Signature Matching

"Signature matching is a way to use built-in information of software components to organize them, navigate through them and retrieve them" (from [Zaremski and Wing, 1995]).

The traditional way to structure software libraries and search through them relies on the distinction of components solely by their name and the directory they are contained in. Though this approach was sufficient for a long time, it no longer is because of the amount of available high quality reusable software components. Thus to yield extra productivity and quality the existence of such a library is not enough anymore. The problem that arises is that programmers often do not find what they need, because the name of a component is not a sufficient means to characterize it. Given the fact that the components over which the searches are done are programming units (like classes, modules, packages, procedures, etc.), additional information is available and can be used to solve this problem.

Signature matching uses specific information about the structure of a software component, namely type information. When looking at a function, rather than trying to guess about its use via its name, one could use the type of function (meaning the list of all the types of its input and output parameters). Signature matching is then the process of determining which library components "match" a query signature. Formally general signature matching can be defined as (according to [Zaremski and Wing, 1995]):

Definition 3 *Signature Match: Query Signature q, Match Predicate M, Component Library C → Set of Components*

Signature Match$(q, M, C) = \{c \in C : M\,(c, q)\}$

This means that the signature (type list) of a function and a component library, which contains a set of software components together with the corresponding signatures, are matched according to a specified matching predicate. The result is a subset of the components contained in C, which "match" (what exactly this means is specified later in this section) the query signature.

Depending on the unit of search Zaresmki distinguishes between function matching and module matching. Module matching is an extension of function matching, as here all functions of a module (i.e. the interface of the module) are matched. As this thesis focuses solely on function matching, only this will be explained further. More information on module/interface matching can be found in [Zaremski and Wing, 1993] and [Zaremski and Wing, 1995].

Function Matching

In function signature matching the search query is based on the function's type and not just its name. This can be seen as a way of using domain-specific knowledge to aid in the search process by exploiting the structure of a function. A type is defined to be either a type *variable* or a type *operator* applied to other types. Type operators are either built-in operators or user-defined operators. Each operator has an *arity* indicating the number of type arguments. Examples for 0-arity operators are the base types like int or bool. The function parameter, indicated by $\rightarrow$, is an example of an operator with 2-arity. A user-defined type, $\alpha\mathcal{T}$, represents a type operator $\mathcal{T}$ with arity 1, where the type of the argument is α.

Two types τ and τ' are equal ($\tau = \tau'$) if either they are the same type variable or $\tau = typeOperator(\tau_1, ..., \tau_n)$,$\tau' = typeOperator'(\tau'_1, ..., \tau'_n)$, $typeOperator = typeOperator'$, and $\forall 1 \leq i \leq n, \tau_i = \tau'_i$.

Given the type of a query τ_q and the type of a function from a library τ_l, the generic form of function matching $M(\tau_l, \tau_q)$ is defined as (according to [Zaremski and Wing, 1995]):

Definition 4 *(Generic Function Match)*

$$M\text{: Library Type, Query Type} \rightarrow \text{Boolean}$$
$$M(\tau_l, \tau_q) = T_l(\tau_l)\, RT_q(\tau_q)$$

where T_l and T_q are transformations (e.g. reordering or renaming) applied

to the library and query types respectively. R is some relationship defined between types (e.g. equality).

Different types of matching can now be defined through the use of different transformations and relationships. Typically matching is divided into exact matching and relaxed matching. Both can be used as matching predicates in the definition of general signature matching and both consider different transformations and relationships.

4.1.1 Exact Matching

"Two function types match exactly if they match modulo variable renaming" (from [Zaremski and Wing, 1995]). The formal definition of exact matching is (according to [Zaremski and Wing, 1995]):

Definition 5 *(Exact Match)*

$$match_E(\tau_l, \tau_q) = \exists \text{ a sequence of variable renamings, } V, \text{ such that } V\tau_l = \tau_q$$

Variable renaming means either the renaming of type variables or of user-defined type operators. This definition shows, that variable names are of no importance regarding signature matching. It shows also that exact match is a very strong limitation, because only renaming is allowed as a transformation.

In Computer Science exact match has its uses, but often does not find all possible answers. This is due to the nature of software, which can sometimes be changed to fit new circumstances. "Relaxed matching" covers this problem and is used to identify possible – but on the first look not perfect – matches.

4.1.2 Relaxed Matching

As said previously exact matching often poses too strong limitations on the result set. It may miss useful functions whose signature are close to the query signature, but that do not exactly match. Slight modifications can be applied to the query signature to yield more matches. These modifications can be classified into two categories: "partial relaxations", which vary the relationship between the types and "transformation relaxations", which vary the transformations applied to the types prior to matching.

Partial Relaxations

Often a more specific query type can replace a very general function type. This is due to the hierarchical nature of types, which are often arranged in a strict hierarchical structure (e.g. classes in C++ or Java). Or it may be difficult for a user to describe the most general type of a function parameter, but a concrete example can be given. Conversely there exist situations in which the user makes queries with a very general type but the library does not contain such a function. It could be possible though, that a function with a more specific type exists, which would be sufficient or could be easily adapted to the needs of the user.

Referring back to the definition of generic function matching, the relationship R between types is equality for exact matching. For partial matches, this relation is relaxed to a partial ordering of the types, based on the generality of types. A type τ is more general than a type τ' ($\tau \geq \tau'$) if τ' is the result of a sequence of variable substitutions applied to type τ. Equivalently, one says τ' is an instance of τ ($\tau' \leq \tau$). One would typically expect functions in a library to be of general a type as possible.

The solution to these problems are "Generalized Match" and "Specialized Match". The formal definitions according to [Zaremski and Wing, 1995] are:

Definition 6 *(Generalized Match)*

$$match_{gen}\,(\tau_l, \tau_q) = \tau_l \geq \tau_q$$

A library type matches a query type if the library type is more general than the query type. Exact match with variable renaming is actually just a special case of generalize match, where all the substitutions are variables, so $match_E \Rightarrow match_{gen}$. This kind of match and the returned results are of special interest, because the user does not need to make any changes to used the functions returned.

Definition 7 *(Specialized Match)*

$$match_{spec}\,(\tau_l, \tau_q) = \tau_l \leq \tau_q$$

This is the converse of generalized match. In fact one can define specialized match by swapping the order of the parameters:

$$match_{spec}(\tau_l, \tau_q) = match_{gen}(\tau_q, \tau_l)$$

In addition exact match is also a special case of specialized match, so $match_E \Rightarrow match_{spec}$

Transformation Relaxations

These types of relaxed matches transform the order or form of parts of the query expression to achieve a match. Examples for these transformations include changing whether a function is curried or uncurried, changing the order of types in a tuple and changing the order of arguments of functions (for functions that take more than one argument).

Uncurrying Functions

Uncurrying is a method from functional programming, where functions with multiple parameters can be described as a series of functions with one parameter each. The complete definition of currying/uncurrying is beyond the scope of this thesis and can be found in [Zaremski and Wing, 1995], page 185, figure 2.

In signature matching uncurrying means the disassembly of the query into its pieces to find a match. Often it is of no importance for the user whether the function is uncurried or curried, because the result of the function application(s) is the same. For example, the uncurried version of a function with three parameters has the type $(\tau_1, \tau_2, \tau_3) \rightarrow \tau$, while the corresponding curried version has the type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau$.

The formal definition according to [Zaremski and Wing, 1995] is:

Definition 8 *(Uncurry Match, Recursive Uncurry Match)*

$$match_{uncurry}(\tau_l, \tau_q) = match_E(uncurry(\tau_l), uncurry(\tau_q))$$
$$match_{uncurry^*}(\tau_l, \tau_q) = match_E(uncurry^*(\tau_l), uncurry^*(\tau_q))$$

Uncurry Match takes two uncurried function types and determines whether their corresponding argument types match. *Recursive Uncurry Match* works similar but allows for recursive uncurrying of the functional arguments of τ_l and τ_q. By applying this transformation to both arguments, the types are transformed into a canonical form. It is not necessary to define a *curry* match, because the uncurry transformation is applied to both the query and library types.

Reordering Types

Oftentimes the order in which the parameters occur is of no importance to the user. One common use of tuples is to group the arguments to a function, where the order is not important. In these cases it can be helpful to change the order to find matches in the supply library. In [Zaremski and Wing, 1995] reordering is defined in terms of permutations. When a function type is given with a tuple as the first argument (e.g., $\tau = (\tau_1, ..., \tau_{n-1}) \rightarrow \tau_n$), a *permutation* σ is a one-to-one mapping with domain and range $1, ..., n-1$ such that $\sigma(\tau) = (\sigma(\tau_1), ..., \sigma(\tau_{n-1})) \rightarrow \tau_n$.

The formal definition is:

Definition 9 *(Reorder Match)*

$$match_{reorder}(\tau_l, \tau_q) = \exists \text{ a permutation } \sigma \text{ such that } match_E(\sigma(\tau_l), \tau_q)$$

This means a library type τ_l matches a query type τ_q if the argument types of τ_l can be reordered so that the types match exactly. This is only possible if both τ_l and τ_q are function types with tuples as first arguments. Applying the inverse permutation σ^{-1} to τ_q would lead to the same result.

These are the concepts of signature matching that are used by this thesis. Further information can be found in [Zaremski and Wing, 1993] and [Zaremski and Wing, 1995].

4.2. Description Logics

Description Logics (DLs) are a family of formalisms for knowledge representation. They are typically used to represent the knowledge of a specific application domain in a structured and well-known way. DLs are thoroughly covered in the "Description Logic Handbook" [McGuinness et al., 2003], this chapter only gives a short introduction to the concepts and theories used in this master thesis.

DLs consist of two parts. A knowledge base and reasoning services. The knowledge base is structured through a description logic language. The basic syntax elements of every description logic language are:

- Concept names:

  These denote atomic "concepts", which describe a subset of the individuals of the used domain.

- Role names:

  These denote atomic "roles", which are used to connect individuals belonging to different concepts. They describe relations between the individuals.

In addition, complex concept descriptions (expressions) can be built from these basic elements inductively with the help of constructors. Constructors for defining concept expressions are specific to each DL. Examples for common constructors are:

- Concept intersection (conjunction): $C \sqcap D$
- Concept union (disjunction): $C \sqcup D$
- Concept complement (negation): $\neg C$

Different description logic languages expand on these standard constructors and add more sophisticated ones. The availability of specific constructors characterizes a specific set of DLs. A very short and often used example is the knowledge base, which consists of the concepts: $male$, $female$ and $person$. With these expression "all women" could be defined as $person \sqcap female$. An equivalent definition would be $person \sqcap \neg male$.

One of the most basic and most used description languages is the family of $\mathcal{AL}$ languages. These allow for the negation of atomic concepts, conjunction of concepts, value/type restrictions and limited existential quantification. The latter two constructs ares used together with concept roles. Roles link two concepts, or more specifically the individuals of two concepts. The example knowledge base might have a $hasChild$ role. This role would link parents and their children. Thus the concept defined by $\forall hasChild.male$ denotes all individuals who have only male children.

This thesis uses some advanced concepts and constructs and thus a more expressive DL is needed. The DL used belongs to the familiy of $\mathcal{ALEN}$ DLs. This is the standard $\mathcal{AL}$ DL extended with *Full existential quantification* (denoted by the letter $\mathcal{E}$) and *Number Restrictions* (denoted by the letter $\mathcal{N}$). This means roles can be combined with concepts using *existential role quantifiers* ($\exists_n R.C$) or *universal role quantifiers* ($\forall R.C$). The former denotes the existence of n roles R where the filler belongs to the set of C and the latter, that every filler for role R belongs to the set denoted by concept C.

The semantics of a description logic is defined by interpreting concepts as sets of individuals and roles as sets of pairs of individuals. Formally the semantic *interpretation* of a description logic is a pair $\mathcal{I} = \left(\Delta, \cdot^{\mathcal{I}}\right)$, where Δ is the *domain* and $\cdot^{\mathcal{I}}$ the interpretation function. Elements of Δ are called individuals, and $\cdot^{\mathcal{I}}$ maps every concept to a subset of Δ and every role to a subset of $\Delta \times \Delta$.

The internal structure of a description logic as described above is called a TBox or taxonomy. It is also possible to specify extensional knowledge within a description logic. In an ABox one can define restrictions on the individuals by using definitions and inclusion assertions. An example of a definition is $women \equiv person \sqcap female$, which denotes that a woman is defined as a female person. An inclusion assertion states to which concept an individual belongs.

A *model* of a TBox is an interpretation satisfying all inclusions and definitions of a TBox.

4.2.1 Inferencing Services

Another important feature of description logics is the ability to perform so called *reasoning* or *inferencing services* over the concepts in a TBox. This enables the description logic to make implicit knowledge visible, that can be extracted from the explicit knowledge introduced into the system.

Two basic reasoning services are offered by most description logic systems, concept *subsumption* and concept *satisfiability*.

Subsumption is typically written as $C \sqsubseteq D$ and answers the question whether, when a TBox $\mathcal{T}$ and two concepts C and D are given, D (the *subsumer*) is more general than C (the *subsumee*) in any model of $\mathcal{T}$. In other words if C always denotes a subset of the set denoted by D.

Satisfiability captures the problem of checking whether, when a TBox $\mathcal{T}$ and a concept C is given, at least one model of $\mathcal{T}$ exists, where C does not subsume the empty concept. In other words, does an interpretation of $\mathcal{T}$ exist in which the set denoted by C is not empty.

4.3. Advanced Inferencing Services

The proposed algorithm for signature matching offers relaxed matching in addition to exact matching, thus non-standard inferencing services are

needed. Concept contraction and concept abduction are used by the algorithm and are introduced here. One of the advantages of both services is that they track the changes that are needed to find a solution for the problem.

To illustrate these concepts the example of an electronic marketplace is used. Here buyers have specific *demands*, while sellers offer their *supplies*. The "simple" goal is to find a *supply* which satisfies the users *demand*. We assume that no *supply* perfectly fits the *demand*.

4.3.1 Concept Contraction

The basic idea of concept contraction is to filter out contradicting expressions. If for example the buyer has a maximum limit of money he wants to pay for a specific service, but the supplier only offers this service for a higher amount, the buyer has to loosen his requirements. He has to *give up* something. This is represented by the concept G while the concepts he can *keep* are represented by the concept K. Formally one takes the supply S and the demand D and checks the satisfiability of their conjunction $D \sqcap S$ w.r.t. the TBox $\mathcal{T}$. The goal is then to retract requirements from D to obtain a concept K such that $K \sqcap S$ are satisfiable w.r.t. $\mathcal{T}$.

The formal definition of a concept contraction Problem (CCP) is defined as (taken from [Colucci et al., 2004b]):

Definition 10 *Let $\mathcal{L}$ be a description logic, S, D, be two concepts in $\mathcal{L}$, and $\mathcal{T}$ be a set of axioms in $\mathcal{L}$, where both S and D are satisfiable in $\mathcal{T}$. A Concept Contraction Problem (CCP), denoted as $\langle \mathcal{L}, S, D, \mathcal{T} \rangle$, is finding a pair of concepts $\langle G, K \rangle \in \mathcal{L} \times \mathcal{L}$ such that $\mathcal{T} \models D \equiv G \sqcap K$, and $K \sqcap S$ is satisfiable in $\mathcal{T}$. K is called a contraction of D according to S and $\mathcal{T}$.*

There exists always the trivial solution $\langle G, K \rangle = \langle D, \top \rangle$ to a CCP. This is the extreme case, that we give up everything of D.

4.3.2 Concept Abduction

After having filtered out all contradictions, it can still happen that a supply does not subsume the demand. This is for example the case if the seller offer includes extra services, which the buyer did not mention in the first place. The solution is to *hypothesize* these additional concepts to the keep.

The formal definition of a Concept Abduction Problem (CAP) is defined as (taken from [Colucci et al., 2004b]):

Definition 11 *Let $\mathcal{L}$ be a DL, S, D, be two concepts in $\mathcal{L}$, and $\mathcal{T}$ be a set of axioms in $\mathcal{L}$, where both S and D are satisfiable in $\mathcal{T}$. A Concept Abduction Problem (CAP), denoted as $\langle \mathcal{L}, S, D, \mathcal{T} \rangle$, is finding a concept $H \in \mathcal{L}$ such that $\mathcal{T} \models D \sqcap H \sqsubseteq S$ and moreover that $D \sqcap H$ is satisfiable in $\mathcal{T}$. We call H a hypothesis about D according to S and $\mathcal{T}$.*

4.4. The RacerPro Reasoner

Racer stands for **R**enamed **A**Box and **C**oncept **E**xpression **R**easoner. RacerPro is a full-fledged knowledge representation system, which allows reasoning over multiple TBoxes. It implements the description logic $\mathcal{ALCQHIR}_+$, also known as $\mathcal{SHIQ}$ (see [Horrocks et al., 2000]). This is the basic $\mathcal{AL}$ logic augmented with qualifying number restrictions, role hierarchies, inverse roles, and transitive roles. RacerPro provides basic reasoning services, which include checking concept satisfiability and concept consumption.

In addition to being a description logic system, RacerPro is also a semantic web reasoning system and information repository. It can manage semantic web ontologies, based on OWL (Web Ontology Language, see [W3C]), and provides sophisticated services like checking the consistency of an OWL ontology or finding synonyms for resources.

RacerPro is available at [rac] including free licenses for research purposes. A Java library for easy access to the systems is available as well, which will be used in the example implementation of the proposed algorithm of this thesis.

The thesis uses RacerPro exclusivly as an example for a description logic system. It uses its basic services for the sample implementation of the algorithm, but will also expand on them with the concepts introduced in this chapter.

4.5. λ Calculus

The λ calculus is a formal system which is related to functional programming languages and is used to analyze function definition, function application and function composition. It was invented in the 1930s by Alonzo Church and

Stephen Cole Kleene and is in itself a programming language.

λ-calculi express both computational and logical information. In λ calculus the basic syntactic element are functions. Every expression stands for a function with a single argument, while an argument itself can be a function with a single argument. The value of the function is also a function. Formally, the λ calculus consists of three elements:

- Variable definition:

  $< expr >::=< identifier >$

- Function abstraction:

  $< expr >::= \lambda < identifier > . < expr >$

- Function application:

  $< expr >::= (< expr >) < expr >$

Another common way is to define the λ calculus through a grammar. When $\mathcal{T}$ is the set of terms and $\mathcal{V}$ the set of variables the following grammar describes the set of terms:

$$\mathcal{T} := \mathcal{V}|(\mathcal{T})\mathcal{T}|\lambda\mathcal{V}.\mathcal{T}$$

The λ calculus is important for this thesis because the "Asset Expression Model", which will be explained in the next section, is based on it.

Simply Typed λ-calculi

The λ calculus in its original form does not know additional restrictions for expression building besides the ones mentioned above. In programming languages it is common to employ a type system to constrain the use of functions. This leads to better efficiency in programming and helps finding some types of programming errors.

The simplest example of a typed λ calculus is the "Simply typed λ-calculus" ($\lambda^{\rightarrow}$). It knows only base types for variables and function types. If $\mathcal{G}$ is the the set of base types, types are given by the following grammar:

$$\tau := \mathcal{G}|\tau \rightarrow \tau$$

Church was the first to develop this explicit type system for lambda calculi.

All variables which appear in λ expressions are assigned to types. Every λ term is annotated with a type. The formal definitions are:

$$\begin{aligned} x \in \mathcal{V} \Rightarrow x \in \Lambda_{\mathcal{T}} & \quad \text{Variable} \\ E_1, E_2 \in \Lambda_{\mathcal{T}} \Rightarrow (E_1)\, E_2 \in \Lambda_{\mathcal{T}} & \quad \text{Application} \\ x \in \mathcal{V}, \sigma \in \mathcal{T}, E \in \Lambda_{\mathcal{T}} \Rightarrow (\lambda x : \sigma . E) \in \Lambda_{\mathcal{T}} & \quad \text{Abstraction} \end{aligned}$$

where $\mathcal{V}$ is the set of variables, $\Lambda_{\mathcal{T}}$ is the set of types, E_1 and E_2 are expressions and $\mathcal{T}$ is the set of terms.

4.6. The Asset Expression Model

This section is based on [Boßung, 2007] and gives a short introduction into the Asset Expression Model and the corresponding concepts that are used by the thesis.

The core of the Asset Expression Model is the "Asset Expression Language" (AEL). It describes pieces of arbitrary medial content through conceptual abstractions. The basic idea is to help to overcome the problems that show up, when people have different contextual knowledge. Throughout our whole life everyone of us gains a unique set of knowledge. When we are confronted with new situations we utilize that knowledge and try to analyze the objects and events at hand. The problem is that some things can be interpreted differently depending on the context knowledge one has. Take for example the painting of a historical battle. An art historian may recognize the depicted scene immediately while a person without such background knowledge sees only people occupied with fighting each other. What the second person now does is to divide the picture into smaller parts. She might for example identify a flag, showing the sign of a specific nation, or recognize the leader of one of the armies depicted. Through this process, of giving individual subcomponents of the picture a meaning, the person finally may be able to identify the picture as a whole. This divide-and-conquer strategy is not uncommon and can be repeated infinitely, until components are identified that are small enough to understand.

The AEL allows to describe arbitrary types of content by identifying subcomponents which are needed to understand the content as a whole and allows to give a meaning to them. These conceptual abstractions are called "Asset Expressions".

4.6.1 Asset Expression Syntax

As mentioned before the AEL borrows the concepts of abstraction and application from the λ-calculus. Indeed a similar syntax is used for the AEs. The only difference is how AEs deal with content, which is shown directly in the expression. Thus the syntactic elements of an AE are:

Content

Any multimedial type of content can in principle be used in AEs. More precisely any representation of medial content. The only limitation is the the medium that is used to present the expression. Paper is for example not very capable of showing videos or playing music (at least not at the time this thesis was written).

Naming Expression

Expressions can be bound to a name. This name can be used in other expressions to refer to the named expression. The syntax for binding an expression to a name is:

$$name := expression$$

Abstraction

The syntax of abstractions is directly borrowed from the λ-calculus.

$$\lambda variable.expression$$

This term means that an explanation is required to understand this expression. The name of this required explanation is "*variable*". The abstraction (as in the λ-calculus) creates a function with one parameter. For example it could be necessary to identify a person in a painting. Multiple abstractions can be used on the same content.

Application

When one has found requirements to identify a piece of content, these can be met by means of application.

$$(expression_1)\, expression_2$$

$expression_2$ (the operator) is applied to $expression_1$ (the operand). $expression_1$ can be an abstraction but it does not have to be. It is said that $expression_2$ meets $expression_1$, while the latter is filled by the former.

Lists

A list of AEs is enclosed in { and }.

$$\{e_1, ..., e_n\}$$

Lists can be used anywhere a simple expression is possible. This is useful for example if the content consists of several expressions (like the scanned pages of a book). Lists can be used as operands as well as operators.

An Example Asset Expression

Figure 4.1 shows an example expression. The content is depicted as a picture and two abstractions are made. "Baum" and "Apfel" are expressions which have been defined before, and which are now applied to the two abstractions.

(λdepicted.(λcarrying. )apple)tree)

Figure 4.1: Sample asset expression

4.6.2 Visual Notation of asset expressions

The visual notation of AEs has illustrational character. As medial content is often not representable in textform a visual notation is helpful . All syntactic elements of AEs can be expressed though visual counterparts. Figure 4.2 shows an example expression. The square boxes in red are abstractions, while medial content is directly shown as the picture. Applications are indicated with greens arrows.

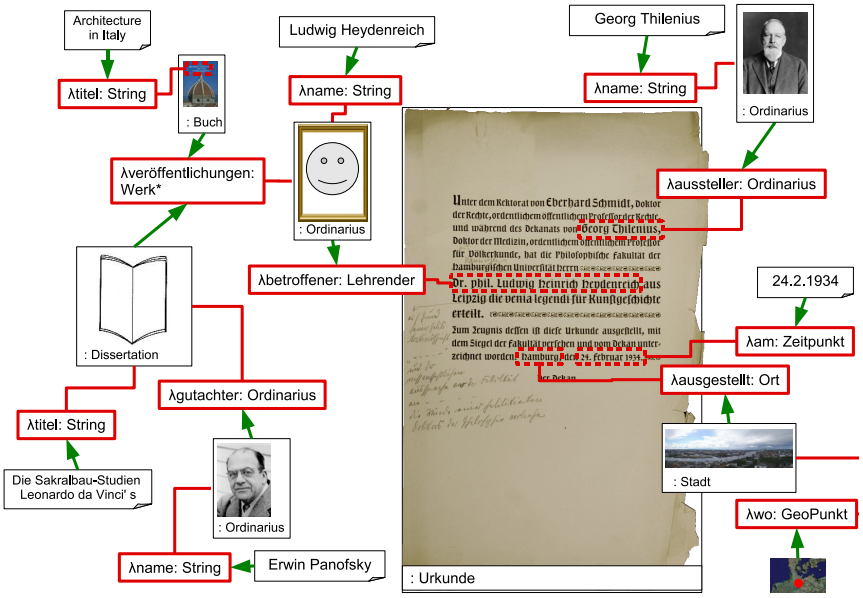


Figure 4.2: Venia-Legendi example classification. (from [Boßung, 2007])

4.6.3 Identifying Content Components

The divide-and-conquer approach of AEs means to identify individual parts of a piece of content. This is not an easy part as medial content can be of many different types. The Asset Expression Model introduces several different selectors for different types of content. These allow for example to specify a region in a picture, or a time sequence in a video. The creation of Asset Expressions is no part of this thesis, thus content components won't be explained in detail here. Further information can be found in [Boßung, 2007].

4.6.4 Typed Asset Expressions

The most important part of AEs for this thesis is its type system. This is because signatures created from AEs will be based on the individual types of every abstraction.

The type system of Asset Expression is based on so called "*Semantic Types*". Semantic types are motivated by the application domain. They are no technical indicators, like types in a programming language, but directly describe semantics of the content. When identifying contents or parts of content the goal is to give a meaning to the analyzed piece. Or in other

words add semantic information to the description about the content.

"Semantic types" are structured in a taxonomy. The most general type – the root – is named the "Any" type. All other semantic types are ordered with the help of a super-type relationship. Every taxonomy is located within a namespace, which acts as the prefix for a type and is used to make type names unique. The local namespace can in addition always be expressed by an empty prefix.

Semantic types are similar to classes in object-oriented programming languages in that they also describe a set of individuals rather than only one instance. It is important to mention that although semantic types are disjunct, content can be classified to be of more than one semantic type. This expresses the view of individual users.

Type Construction

Semantic types can be simple types, function types or list types. The first ones are embedded in the type hierarchy, the second ones are constructed from two semantic types by using the $\rightarrow$ type constructor. This construction can be applied repeatedly:

$$T_1 \rightarrow T_2$$

or

$$(T_1 \rightarrow T_2) \rightarrow T_3 = T_1 \rightarrow T_2 \rightarrow T_3$$

Types are separated by a colon from an expression:

$$e_1 : T$$

As said before types are ordered by a super-type relationship. Every type except the "Any" type have a single super-type. This relationship is written as $<:$, e.g.:

$$T <: S$$

where T is more specific than S.

Expression Typing

Expressions are typed by adding types to variables and multimedial content. The new syntax of abstraction and content are:

$$\lambda v : Type.e$$

for a variable v and an expression e and

$$C : Type$$

for a content C.

Figure 4.3 shows an example typed expression. The fruit carried by the tree is an apple. Thus the picture depicts an appletree. The abstraction has the function type: $Fruit \rightarrow Appletree$. It is also possible for the user to

Apple:=  :fruit

(λcarrying:Fruit.  :Appletree)Apple

Figure 4.3: A types asset expression

change the type of an expression to a more specific one. This is called lifting and has several applications. One being the import from a more general domain, which was not modeled as specifically as the new one.

The main goals achieved through the typing system are:

- Filter out non-sensical expressions:

  Applications are only allowed if the type of the abstraction and the type of the expression applied match.

- Capturing the nature of large expressions as a whole:

  This allows for a classification of content, because the type of the content is independent of the number of explanations employed.

- Relate expression which describe similar real-world entities:

  This also helps in building a classification system for content, as similar real-world entities are described by the same semantic type.

Typing Rules

The typing rules follow examples from functional programming languages an can be found in detail in [Boßung, 2007].

Chapter 5. The Solution

Summary. *This chapter describes the developed signature model and the signature matching algorithm. It gives an overview over how concept contraction and concept abduction are employed to find relaxed matches. In addition different classes of penalty functions are analyzed.*

5.1. Development Of A Signature Model

The basic idea behind a signature model is to make the inner structure and semantics of a piece of medial content visible. Thus its core function is to represent the structure of a signature in a machine readable (and understandable) way. To reach this goal a description logic system is used. The signature model developed in this thesis is based on the considerations made in [Boßung, 2007].

Description logics allow to manage and manipulate a knowledge base. As described in detail in the introductory chapters, this knowledge base usually consists of a terminology (TBox) and assertions (ABox). When dealing with the storage of signatures an ABox is not necessary and thus not used. In fact signatures resemble concepts of a TBox, as they also describe sets of individuals, rather than one specific individual.

The first step to create the signature model is to introduce the types of the used context into the TBox. This is done by creating a subsumption hierarchy over the types. Every type A has a relationship to its super-type S:

$$A \sqsubseteq S$$

No two semantic types are identical, which means appropriate disjointness rules have to be applied to each type A:

$$A \sqsubseteq \neg T_i$$

where T_i are all the siblings of A.

Figure 5.1 shows a part of such a hierarchy. In this taxonomy “Fisch” and “Fleisch” are sub-types of the “Nahrungsmittel” type and thus are disjoint. When developing a signature model several other things have to be taken

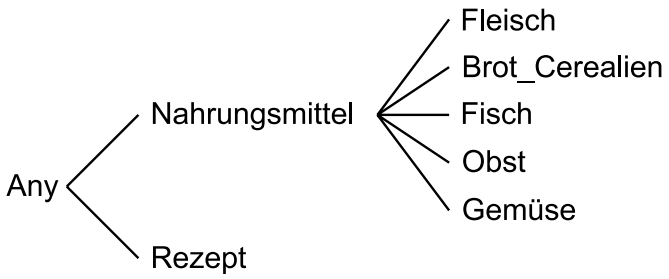


Figure 5.1: The first levels of the food taxonomy

into account. The domain of object oriented programming is used as a starking point, because signature matching is already successfully employed there. A look at how a (function-) signature is designed there helps to identify the basic needs (a Java function signature is taken as an example, but the result would be the same for any programming language).

```
String aFunction(int parameter1,
                 int parameter2, boolean parameter3)
```

Looking at this signature one can identify several characteristics about a signature:

- There are two types of parameters: input (parameter1-3 in the example) and output (of type "String" in the example; also called return-type in Java) parameters.
- The total number of parameters a signature has (including (four) or excluding (three) the return-type).
- The number of parameters of different types (three in the example, because parameter1 and parameter2 are of the same type).
- As parameter names are irrelevant and only parameter types are analyzed, parameter1 and parameter2 can be summed up. This is done by introducing a type's cardinality, meaning the number of occurrences of a signature.

The goal is to identify and define the structure of an arbitrary piece of content through the signature. Taking the characteristics identified above into account, the following aspects are of special importance to the signature model:

- Cardinality:

  Because types are organized in a type hierarchy, it is necessary to think about what it means for the cardinality of a type, if some of its sub-types are part of the same signature. The super-type relationship expresses that the set of individuals depicted by the super-type includes every individual of the set depicted by its sub-types. This means a sub-type always fulfills the requirements of its super-type. Thus the calculation of the cardinality of a type has to take all its subtypes into account. Formally the cardinality of a type T is defined as $card_T = c_d + c_s$, where c_d is the number of parameters of T in the signature and c_s is the number of parameters in the signature which are of the (transitive) sub-types of T.

- Restrictions on the return-type:

  The return-type expresses the actual classification of the piece of content, which means that every piece of content needs to have a return-type. As semantic types are disjunct it is not allowed that a piece of content which is described by only one signature has more than one return-type. Thus every signature contains exactly one return-type.

- Restrictions on the number of parameters:

  The signature matching algorithm matches input types. It does thus not make sense to take pieces of content into account which are described by a signature that contains no parameters, i.e., which have no inner structure. This is why the model requires that at least one parameter is present in each signature.

Taking these points into consideration a general signature concept can be developed, which is valid for every signature:

$$sig \equiv \exists_{\geq 1} hasParameter.Any \sqcap \exists_{=1} hasReturnType.Any$$

Concrete signatures are then modeled as specializations of this concept. Each parameter T is represented in the signature by two conjuncts:

$$\exists hasParameter.T \sqcap \exists_{=n} hasParameter.T$$

The first conjunct describes the type of the parameter and the second the cardinality of the parameter.

Signatures can also be described as compact graphs. Nodes represent the types of the parameters, which are annotated with the cardinality. The connections between the nodes describe the subsumption relationship of the type hierarchy. Figure 5.2 shows the graph of a signature containing two parameters in total, one being of type "Pork" and one of type "Fruit". One

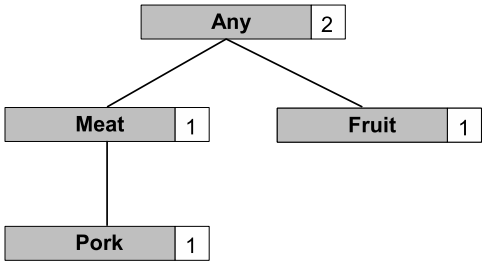


Figure 5.2: Example signature graph

can see in this example, that the supertype of the type "Pork" – which is "Meat" – is also present. As it has the same cardinality as "Pork" no parameters of the type "Meat" are part of the signature. It represents the fact, that a parameter of type "Pork" can also be used as parameters of type "Meat".

The last important point is related to the semantics of a "content-signature". The signature over a content expresses the things necessary to understand this specific piece of content, but this does not mean, that other information is contained within the content. This means there can be additional information contained in the content which is not part of the signature. Thus in addition to the direct model for a signature, a "callability model" has to be defined. This is necessary, because to call signature with n parameters of type T, a calling signature needs "at least" n parameters of type T. Additional parameters of that type or parameters of a type that is not (yet) present in the signature do not matter. The callability is represented in laxer restrictions on the cardinalities of the signature.

$$\exists hasParameter.T \sqcap \exists_{\geq n} hasParameter.T$$

are thus the conjuncts for a parameter of type T in the callibility model of a signature.

5.2. The Matching Algorithm

The second part of the solution is the signature matching algorithm for relaxed signature matching. The following sections give detailed information about the decisions made throughout development.

5.2.1 The Signature Matching Facility

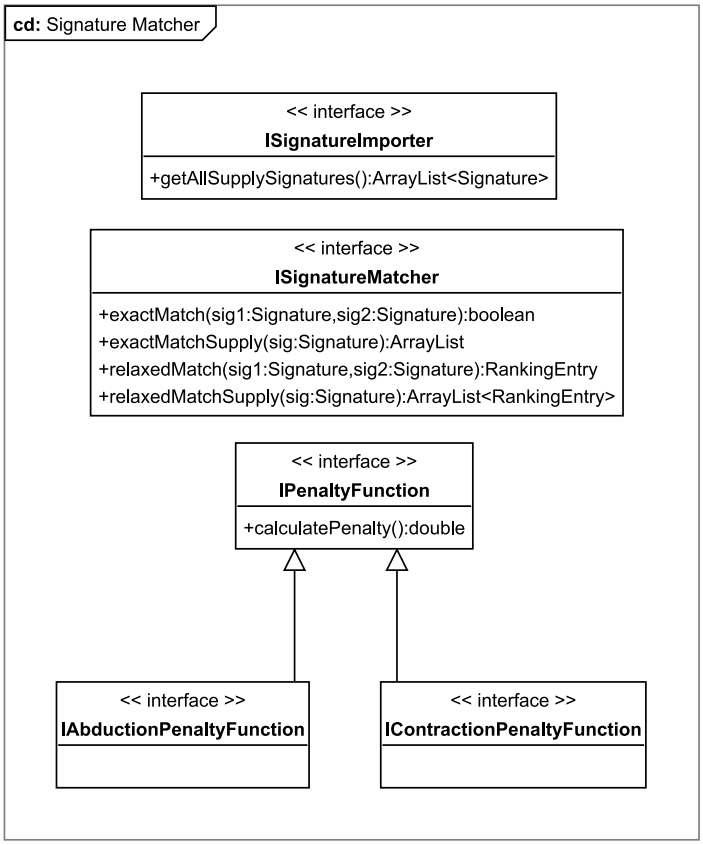


Figure 5.3: The signature matcher class model

Next to the signature model, the biggest achievement of the thesis is the functionality to match these signatures. This part is again based on the considerations on signature matching in [Boßung, 2007]. The signature matching facility consists of three parts. Importer functionality to load signatures from a description logic system. Signature matchers, who perform the actual matching and penalty functions, which are used by the signature matchers to

assign a ranking to the available supply signatures. Figure 6.3 shows the general class diagram of the facility. `ISignatureImporter` is the interface of the signature matching facility. The only method `getAllSupplySignatures()` is used to import the supply signatures into the system. The `ISignatureMatcher` interface describes the basic methods for matching signatures. The penalty functions for CC and CA have their individual interfaces to account for specific differences in the way they calculate the penalties. They inherit though from the same basic penalty function interface, which specifies the `calculatePenalty` function.

5.2.2 The Signature Matcher

The signature matcher matches demand signatures with the callibity models of a set of supply signatures. It offers three different matching methodologies. Exact-Matching, Generalized-Matching and Relaxed-Matching.

Exact- and Generalized-Matching

Two signatures match exactly, if all their parameters are pairwise equal (see section 4.1 for the definition of equality of types). This standard approach to exact matching was not sufficient for the used signature model. Due to the callability model for supply signatures, exact matching does not deliver all "exact" matches. As explained previously, the callability model refers to the fact that additional information does not matter and is not taken into account when searching for a perfect match. In terms of a description logic this means, that not only exact matches are perfect matches, but also specializations of the supply. To take this into account generalized matching is also part of the design. Generalized matching requires to check whether a demand signature is subsumed by a supply signature. This is a case where the optimized uses the subsumption relationship for matching signatures.

Relaxed-Matching

The relaxed matching algorithm is more complex than one for exact matching for two reasons: Firstly the algorithm consists of three different steps that have to be done and secondly different penalty functions can be applied throughout the algorithm. The algorithm utilizes both concept contraction and concept abduction to identify missing and contradicting information in the demand. To make the evaluation of different penalty functions possi-

ble the corresponding interfaces that were introduced previously are used to allow for the easy exchange of the penalty functions. For contraction and abduction different penalty functions are used, which allows for a greater flexibility when trying to achieve better(meaning more realistic) ranking results.

5.2.3 The Running Example

To clarify the algorithm a running example is used in the following chapters. Figure 5.4, 5.5 and 5.6 show a demand and two supply signatures which are compared against each other. The demand signature has two parameters,

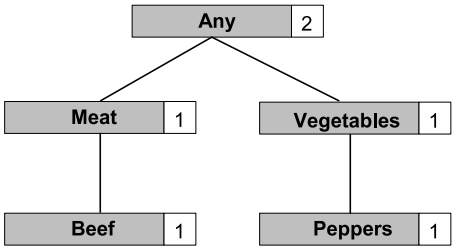


Figure 5.4: An example demand

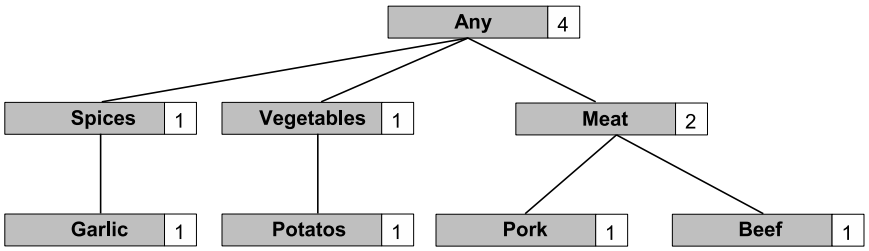


Figure 5.5: Example supply S_1

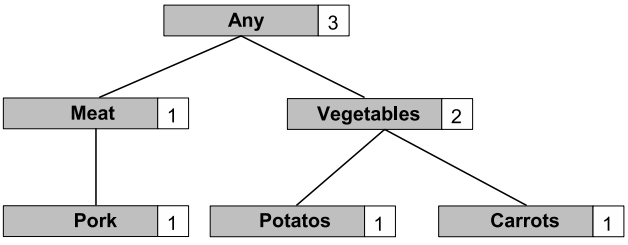


Figure 5.6: Example supply S_2

while supply S_1 has four and supply S_2 has three. What can be seen on the pictures is that both supply signatures do not match the demand perfectly. For example supply S_1 requires two parameters of type "Meat", while the demand only offers one. Supply S_2 on the other hand requires only one parameter of type "Meat" but it further specifies the parameter to be of type "Pork", which does not exist at all in the demand. The following two sections show how concept contraction and concept abduction filter these inconsistencies out and assign corresponding penalties.

5.2.4 Concept Contraction (Inconsistency)

Concept contraction for this scenario is very restricted because of the chosen signature model. Contradicting information is only present in the cardinality of parameters. In addition due to the callability model, it is only possible that the demand contains not enough parameters of a certain type and not too many. Thus the core mechanics for concept contraction are rather simple and are shown in the following pseudo-code excerpt:

```
FOR each parameter in demand signature except Any parameter
IF demand parameter type exists in supply parameter list
  IF supply parameter cardinality>demand parameter cardinality
    calculate penalty
  ENDIF
ENDIF
Calculate penalty for Any parameter
```

Each parameter pair is checked for inconsistencies, which means in this case that the demand offers a lower number of parameters of a specific type than then supply needs. This is then corrected and a penalty according to a penalty function is added.

A unique case is the "Any" parameter. This is the topmost parameter in any given type-hierarchy. It is the only parameter that has no parent and only children. The "Any" parameter does not only represent the number of parameters of the type "Any" but also the number of parameters of the whole signature. A problem arises here because of the additional information that is not taken into account when using the callability model. Thus if the "Any" parameter would be compared just as it is, it would lead to an unsatisfiable result which might also lead into an unsolvable contraction.

The solution used is to take only the relevant parameters of the demand into account, meaning only those parameters whose types are reflected in the supply. This makes sense because although additional parameters do not count toward penalties, they count toward the satisfiability of a signature in the TBox. Figure 5.7 shows the contractions done for the sample signatures. Supply S_1 requires two parameters of type "Meat" and at least four parameters altogether. Supply S_2 requires two parameters of type "Vegetables" and three parameters altogether. In both cases the "Keep" of the contraction process is created by copying the demand signature and changing the cardinalities accordingly. In addition penalties are calculated based on the changes made (see section 5.2.6).

5.2.5 Concept Abduction (Incompleteness)

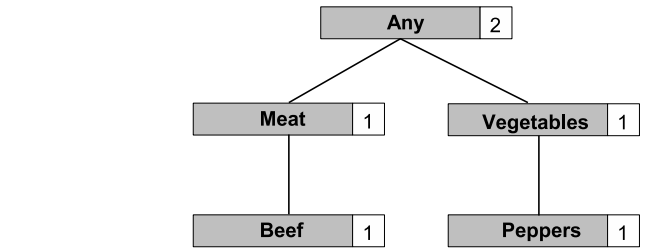
After concept contraction is applied there is still the chance, that the demand is not able to "call" a supply signature. This is because of missing information in the demand, rather than contradicting information. When one looks at the example signatures one can see for example that supply S_1 requires one parameter of type "Spices" which the demand does not contain. Concept abduction covers this problem.

Because of the used signature model missing information only relates to missing parameter types in the demand. Cardinality conflicts were detected and fixed during the contraction phase and thus do not play a role anymore. Concept abduction creates a hypothesis which has to be added to the demand so that it is subsumed by the supply. This also adds flexibility regarding penalties and means only new parameters are added and the penalty is calculated according to a penalty function.

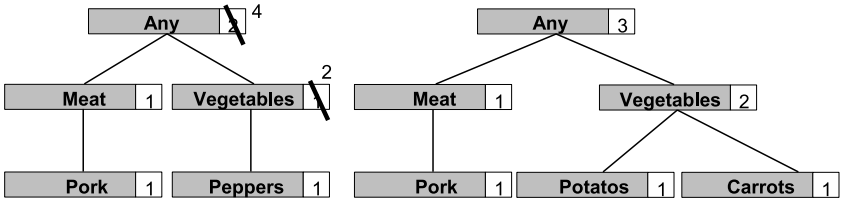
The following pseudo-code excerpt shows the basic flow of the abduction algorithm.

```
FOR each parameter in supply signature
IF supply parameter type does not exist in Keep's parameter list
  Add parameter to Keep
  Calculate corresponding penalty
ENDIF
```

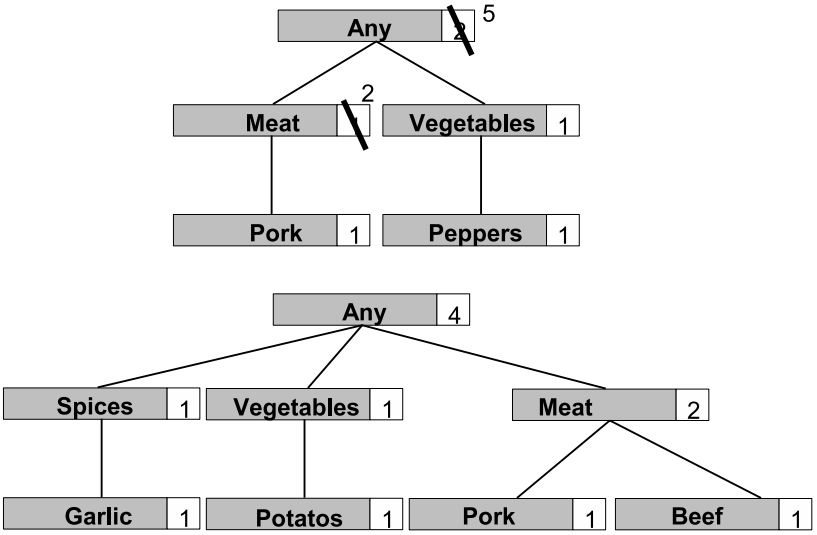
Special attention has been paid to the step where the parameter is added to the Keep's parameter list due to the used signature model. Parents



(a) Demand



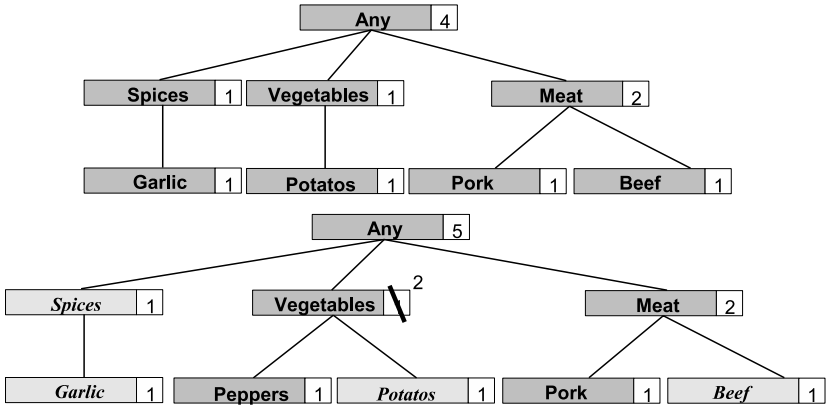
(b) Keep and Supply S1



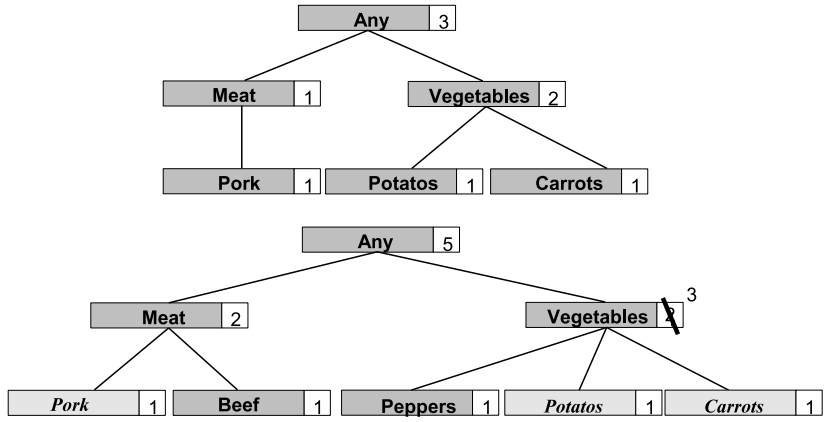
(c) Keep and Supply S2

Figure 5.7: Two examples of concept contraction. In both examples the demand does not have enough parameters of one category and the total number of parameters is wrong

and/or children of the parameter type to be added have to be taken into account. Section 6.2.7 gives detailed information on this. Figure 5.8 shows the hypothesis generated for the two example supplies. In case of supply S_1 parameters of type “Garlic”, “Potatoes” and “Pork” are added, each with



(a) Keep and hypothesis of supply S1



(b) Keep and hypothesis of supply S2

Figure 5.8: Two examples of concept abduction. The missing concepts are added and the cardinalities are propagated up to the Any concept.

cardinality one. For supply S_2 parameters of type "Pork", "Potatoes" and "Carrots" have to be added. What is important here, is that the cardinality of the parameters for "Vegetables" and "Any" have to be fixed to match the total number of parameters of the signature. This is because the demand contains parameters, which are not required by the supply.

5.2.6 Penalty Functions

Penalty functions have the task to quantify the differences found between a demand and a supply signature. During both phases – contraction and abduction – these penalty functions are used to assign a ranking to the found

alternative. To be as flexible as possible, and to be able to evaluate the effectiveness of different penalty functions, a set of interfaces for these functions has been designed. A sub interface for each class of penalty functions was added and is used by the corresponding part of the algorithm. The single method `calculate()` is defined in the super-interface and calculates the penalty of an action according to the given method-parameters.

The penalty functions can be roughly classified according to the type of information they take into account. The first class just looks at the cardinality, both the cardinality of the added/modified parameter and the whole demand-signature. The second class takes also specialization into account. This refers to the fact that an added parameter with a very specific type restrains the signature much more, than an added parameter with a very generic type. The third class looks at the number of different parameter types contained within a signature. This is different to looking at cardinalities, because the information added by a parameter of a new type to a signature is different from the information added by a parameter of a type, which is already contained in the signature.

The signature matching algorithm is tested with all three classes and a fourth class, which combines all properties of the other three classes.

1. Class: Cardinality

This type of penalty functions can be applied to both contraction and abduction. In contraction simply the difference between the cardinality of the specific type in the demand and supply signature are calculated. In abduction the cardinality of the parameter added and the cardinality of the whole demand signature are compared. Both calculations can be configured through the application of weights. The following formulas show possible implementations of penalty functions of this class.

$$\Pi_c = c_c * (card_s - card_d)$$

$$\Pi_a = c_a * card_{np}$$

where c_c and c_a are constants, $card_s$ and $card_d$ are the cardinality of the specific type in the supply and the demand signature respectivly and where $card_{np}$ is the cardinality of the type hypothesized during the abduction phase.

2. Class: Specialization

The second type of penalty functions, checks the depth of a type in the semantic type hierarchy. The assumption made is that a more specific type results in a smaller set of individuals contained in the resulting concept. This may not always be true, because it is dependent on the design of the domain. Some parts of the hierarchy may be designed much more detailed than others, which can break this assumption. Nonetheless these penalty functions assign a penalty depending on the depth of the parameter added. A slight modification is to take the maximum depth of a signature into account. The following formula shows how the penalties are calculated.

$$\Pi_a = depth_{np}$$

$$\Pi_{a1} = depth_{np}/depth_d$$

$depth_{np}$ is the depth of the new parameter type in the type hierarchy and $depth_d$ is the maximum depth of all parameters of the demand signature.

3. Class: Semantic-type

This type of penalty functions only looks at the semantic type of a parameter added. It can not be applied to concept contraction, because here only the cardinality of parameters is changed. The underlying assumption of these penalty functions is that with the addition of new type restrictions one restricts the number of individuals specified by the signature. The algorithm compares the number of parameter types of a signature with the types added through abduction.

$$\Pi_a = 1/number\_of\_types_d$$

4. Class: Mixing

The fourth and final class analyzed is a mixture of all classes mentioned above. The idea is to take as much information as possible into account when calculating the results. This formula can be modified by changing the weights of the different parts. The formula takes the variable from the classes above and composes them to a complex total.

$$\Pi_c = (card_s - card_d)/2 + depth_s * (card_s - card_d)/2$$

$$\Pi_a = card_{np}/2 + depth_{np}/depth_d$$

Chapter 6. The Prototype

Summary. *To test the algorithm a proof of concept application was developed. This chapter gives insight into the decisions made during the development. It also highlights problems related to the chosen description logic system and solutions used to overcome these.*

6.1. Design

The design of the prototype is straightforward. According to the chosen sample domain a signature model based on the general model developed in the last chapter (see 5.1) was created. Based on this model a signature matching framework was then designed and implemented using the RacerPro description logic system. It employs the contraction and abduction algorithms presented in the previous chapter. The following chapters give insight into the decisions that had to be made throughout the design process.

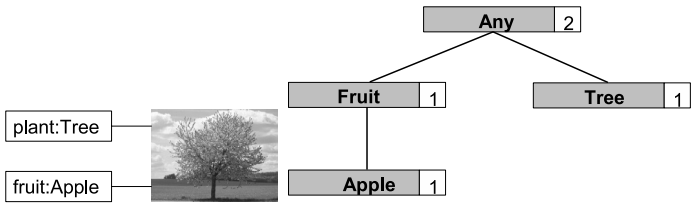
6.1.1 The Asset Expression Signature Model

Asset Expressions (AEs) describe content with the help of abstractions (see chapter 4). The combination of abstractions and applications are seen as explanations of specific parts of the content, which are needed to understand the content. The signature model described in the previous chapter only uses type information of parameters and thus variable naming and the ordering are not taken into account. This means that the signature model based on AEs uses only the type information of abstractions. The signature model used is identical to the one developed in the last chapter.

Figure 6.1 shows an example AE, the hierarchy of parameter types and the signature concept according to the signature model. The signature expands on the general concept and adds two parameters, one of type “Apple” and one of type “Tree”.

6.1.2 Description Logic System Or Object Model?

It was decided to use an object-oriented approach for the sample implementation. This was done mainly for two reasons. Firstly a Java library to access the RacerPro reasoner is available and secondly object-orientation



(a) Asset expression (b) Type hierarchy

$$sig_A \equiv sig \sqcap \exists_{=2} hasParameter.Any \sqcap \exists_{=1} hasParameter.Fruit \sqcap \exists_{=1} hasParameter.Apple \sqcap \exists_{=1} hasParameter.Tree$$

(c) Signature concept

Figure 6.1: Example of an Asset Expression modeled in a description logic concept

offers several comfortable features which help when implementing the algorithm. The type tree for example can be very well modeled through classes and by using a hashmap for storage the access is also very efficient.

One important question is then when to use services of the description logic system and when to work only with the object representation of a signature and the type tree. Some functionality is only available through the description logic system, like checking the subsumption relationship, while some is not supported by it. One example where working with the description logic system is faster, is when the equality of two signatures is checked. Per definition two signatures are equivalent, if their parameters are pairwise equal (meaning having the same type and cardinality). Implementing this with the object model would result in a lot of unnecessary comparisons. The description logic system is optimized for these types of queries and works very efficiently. On the other hand, if only semantic types are to be compared (eg. during contraction and abduction), working on the object model proves to be more efficient. As the names of semantic types are unique it is not necessary to call the description logic system (which would include setting up sockets, transmitting the date, calculation of the result by the

description logic system, and transmission of the result). It is sufficient to check for string equality of the type representations in the model.

6.1.3 Modeling The Type Hierarchy

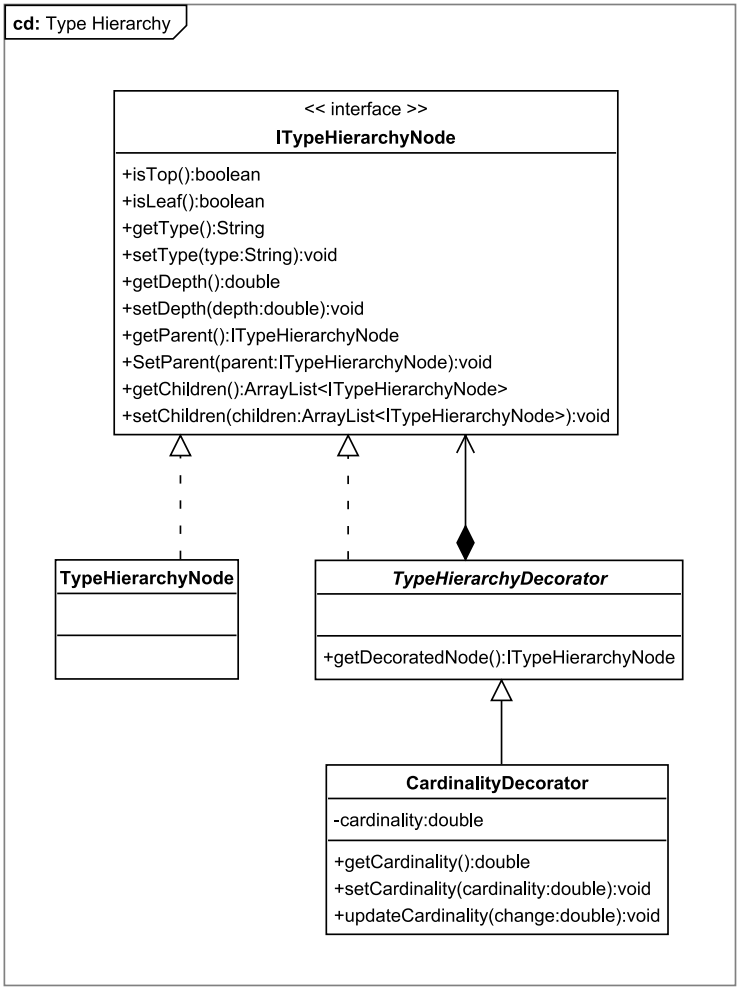


Figure 6.2: The modeled type hierarchy and the employed decorator pattern for cardinalities

The type hierarchy of the semantic types can be quite big, thus efficient means are needed to store and access it. As the whole hierarchy is often needed during several parts of the algorithm, an object representation of the type hierarchy was designed. When deciding on the structure of the representation two things about the hierarchy were the most important.

Firstly it is often necessary to find the parents and/or children of a type, meaning the object representation had to mirror these relationships. This is mostly necessary due to the subsumption relationship, which makes types representable by others. Secondly it has to be easy to manage the types of a signature and annotate them with cardinalities. This led to the decision to represent the types in a tree-like structure and at the same time store every type in a hashmap. The string representation (or name) of the type is used as the key (because of its uniqueness), while the object representation of the type is the value. The hashmap brings two distinct advantages. Naturally it is very fast to access an arbitrary type. In addition the value objects of a hashmap can be accessed as a Java collection. This makes it possible to use iterators instead of recursive methods, which speeds up the algorithm considerably.

In addition to the representation of the type hierarchy it is also necessary to represent the parameter lists of signatures. This was designed by applying the decorator pattern to create a cardinality decorator for each type. Figure 6.2 shows the corresponding interfaces and classes.

Types are represented through the `TypeHierarchyNode` class. For every parameter of a signature is then decorator object created, which adds the cardinality to the type. Both, the semantic types of the type hierarchy as well as the types of the parameters of a signature are stored in a hashmap. Through the decorator it is always possible to get the original type object and thus the corresponding parent/child relationships. This design has also a very low memory consumption, because the original type tree exists only once and only lightweight decorator objects are created for each signature.

6.1.4 The Signature Matching Facility

The signature matching facility consists of three parts. 1. Importer functionality to load signatures from a description logic system. 2. Signature matchers, who perform the actual matching and 3. penalty functions, which are used by the signature matchers to assign a ranking to the available supply signatures. Figure 6.3 shows the class diagram of the facility. The importer and penalty function interfaces have already been described in the precious chapter (see section 5.2). The signature matcher interface is described in detail in the following section.

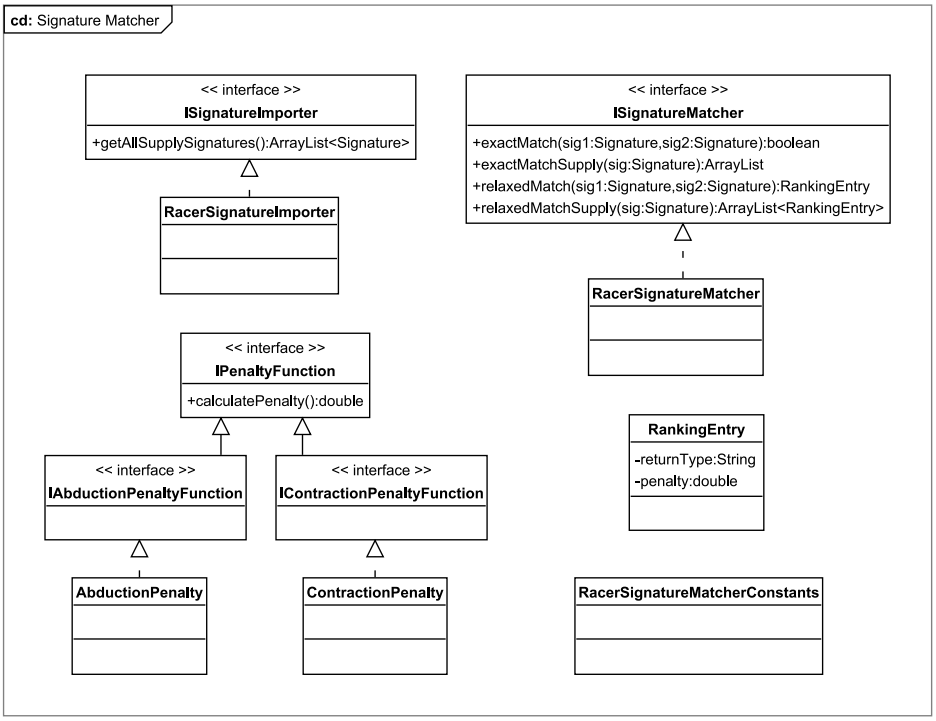


Figure 6.3: The signature matcher class model

The Signature Matchers

The signature matchers are described by the `ISignatureMatcher` interface. It contains five functions:

- `exactMatch(Signature, Signature)`

  Matches the two given signatures. Returns true or false.

- `exactMatchSupply(Signature)`

  Matches the given signature with all available supply signatures. Returns a list of signatures that match the given signature.

- `generalizedMatch(Signature, Signature)`

  Matches the two given signatures using the subsumption relationship. Returns true or false.

- `generalizedMatchSupply(Signature)`

  Matches the given signature with all available supply signatures. It employs the subsumption relationship and returns a list of signatures that match the given signature.

- `relaxMatchSupply(Signature)`

  Matches the given signature with all available supply signatures. Returns a list of ranking entries, which contain the return-type (the classification) of each supply signature and the assigned penalty.

The general matcher interface should be as generic as possible, to allow for a broad range of implementations. Together with the IDLHelper and ISignatureImporter interfaces it offers a great flexibility regarding the decisions that can be made for the actual implementation.

6.2. Implementation

The implementation of the prototype was done in Java, using a proprietary Java API to access the description logic system. The following sections give a short overview over the implementation and the working algorithm.

6.2.1 The Object Model For Signatures

After the signature model was developed the next step was to translate it into the object world. The model developed states, that every signature consists of at least one parameter and a return-type. Figure 6.4 shows the corresponding class diagram. As mentioned previously, the actual parameters of a signature are stored as decorated type objects in a hashmap.

6.2.2 Creation Of The Type Hierarchy

The type hierarchy has to be created during the initialization phase of the algorithm. The class responsible for this is the description logic helper. Upon the creation of the class the `createTypeTree` method is called. It creates the "Any"-Type as this type is part of every type hierarchy, and then calls the `initializeHashMap` method, which recursively adds all other types to the hierarchy. This is the first and only time that the RacerPro system is called to access the type hierarchy. As said before the types are

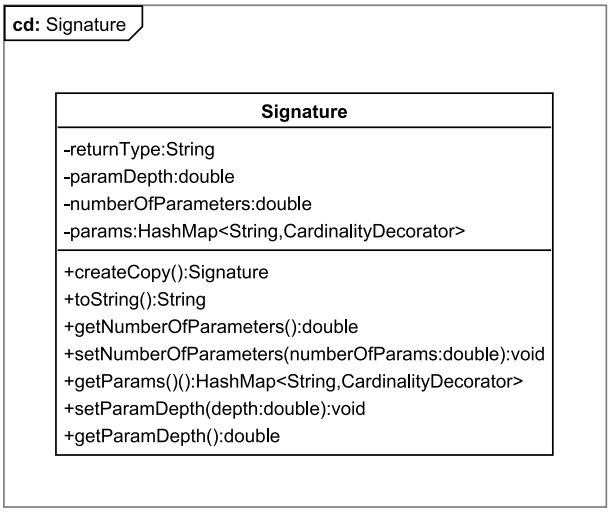


Figure 6.4: The signature class

created with parent and children relationships properly set up and are then added to a hashmap.

6.2.3 The Signature Importer

The signature importer loads signatures from the description logic system. As the signatures are stored as concept definitions in the description logic system, the return values from the system are strings. These strings are composed of conjuncts of parameter definitions and their corresponding cardinality definitions. The task of the signature importer is to convert these strings into object representations. It does so by filtering the parameter types and cardinalities out of the concept string. With the help of the `CardinalityDecorator` class an object representation of each parameter is then created. The `CardinalityDecorator` contains a reference to the type object in the type hierarchy as well as the cardinality of this parameter in the signature.

The signature importer is also used to get the library of supply signatures. Because of this the main method described by the ISignatureImporter interface is `getAllSupplySignatures()`. It gets the list of names of all supply signatures by utilizing the fact, that all supply signatures are subsumed by the general signature concept. This allows to easily identify all supply

signatures, without knowing each specific name.

6.2.4 The Description Logic Helper

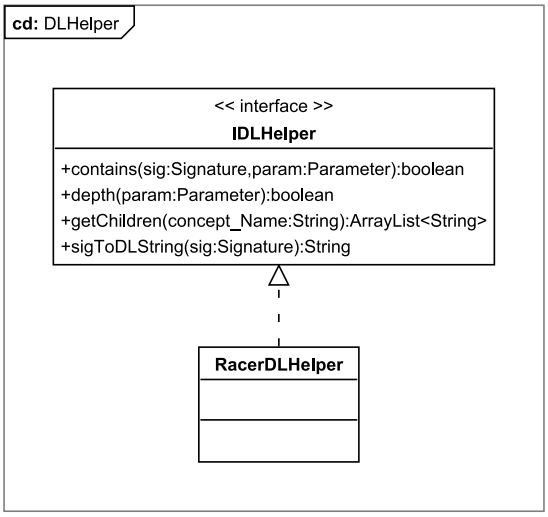


Figure 6.5: Description logic system helper class

The signature matcher uses the description logic system on several occasions. A description logic helper class was created, which encapsulates all advanced functionality related to the description logic system. It also holds the reference to the hashmap containing the type hierarchy and thus is a central access point for the algorithm. Figure 6.5 shows the interface and the corresponding RacerPro implementation class. The helper class offers the following functionality:

- "contains": This method checks whether a parameter is contained in a signature or not.
- "depth": This method calculates the depth-level of the type of the given parameter in the type hierarchy.
- "getChildren": Retrieves the list of all children of a given concept.
- "sigToDLString": Helper method to generate a description logic system compatible concept definition from a signature object

- "getTypeTree": Returns the hashmap, which contains the type tree. This is initialized in the constructor of the class.

6.2.5 The Signature Matching Algorithm

The signature matching algorithm is based on [Boßung, 2007] and [Zaremski and Wing, 1993]. As explained in the precious section it offers two "flavours" of signature matching: exact matching and relaxed matching. While the overall structure of the algorithm is based on the algorithm by Zaremski, the actual comparison methods are – where possible – directly taken from the RacerPro system. The following sections give a detailed insight into the two developed algorithms.

Exact Matching

The task of the exact matching algorithm is to find perfect matches to a given signature. The algorithm algorithm offers two methods for this:

```
exactMatch(Signature, Signature)
```

```
exactMatchSupply(Signature)
```

The first method compares two signatures and returns true or false accordingly. The algorithm uses a helper method to create a valid concept string out of each signature and then uses the RacerPro functionality to compare the two signatures. The second method compares the given signature to all supply signatures present in the system. The following pseudo code shows the basic implementation of these methods:

```
Retrieve all supply signatures
IF number of parameters of signatures are different THEN
  RETURN false
ENDIF ELSE
  Create description logic conform signature strings
  Call equivalence functionality of description logic system
  RETURN evaluation of result
```

While this method yields exact matches in terms of concept comparisons, it does not yield all exact matches for the proposed signature model (see section 5.1). This is because the callability model allows for a perfect match

for every signature, which is subsumed by the supply signature. To ensure that all matching signature are found, a generalized matching algorithm was developed. Instead of trying to match signatures exactly, it uses the subsumption service of RacerPro to find possible matches. This leads to a much larger result set, but nonetheless only to signatures which fit the demand perfectly.

The generalized matching facility offers also two methods: One for comparing two signatures and one for comparing a signature to the whole supply.

```
generalizedMatch(Signature, Signature)
```

```
generalizedMatchSupply(Signature)
```

Relaxed Matching

The relaxed matching algorithm is implemented similar to the exact matching algorithm. The employed description logic system is used for every logical comparison possible, with exceptions to improve the performance. As explained in earlier sections the call of the RacerPro system is very complex and not very performant. Figure 6.6 shows the activity diagram and thus the overall programming flow of the algorithm. It consists of three parts:

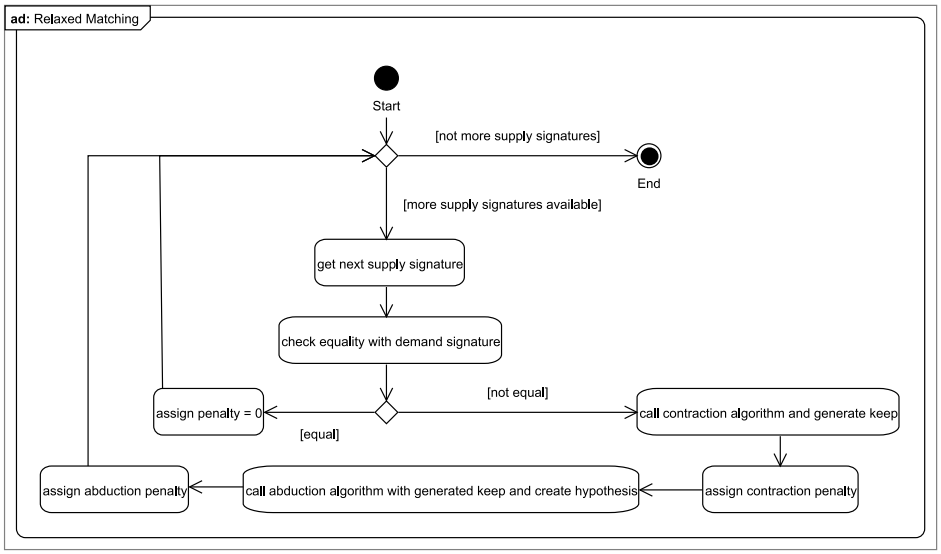


Figure 6.6: The flow of the relaxed matching algorithm

1. Finding exact matches

2. Using contraction to filter out contradictions and generate the keep

3. Using abduction to hypothesize missing information

Exact matching has already been described. The other two parts are explained in detail in the next sections.

6.2.6 Concept Contraction

The concept contraction algorithm is based on the definition presented in chapter 4, but several adjustments had to be made, so that it fit the signature

Concept contraction is used to filter out contradicting requirements of demand and supply. As mentioned in section 5.2.4 the "Any" parameter takes a special role. The algorithm needs to account for the fact that also the parameters of the demand, that are not relevant for the supply, are reflected in the cardinality of the "Any" parameter. This is done by taking only the parameters into account that appear also in the supply. This is done via the `getUnrelevantParameters` method, which returns the number of parameters that appear in the demand, but not in the supply.

After this the rest of the parameters are checked for contradictions. The following pseudo-code shows the operation of the algorithm.

```
Create copy of demand signature parameters
Calculate penalty for Any parameter
FOR each parameter in demand signature except Any parameter
IF demand parameter type exists in supply parameter list
  IF supply parameter cardinality > demand parameter cardinality
    demand parameter cardinality = supply parameter cardinality
    calculate penalty
  ENDIF
ENDIF
```

The algorithm first creates a copy from the demand signature to be used as the keep. It then takes every parameter from the keep, that is relevant for the supply, and checks whether the keep parameter has at least the cardinality of the corresponding supply parameter. If this is not the case, the cardinality of the parameter in the keep is set to the cardinality of the supply parameter. A penalty is then calculated according to the given

penalty function, based on the old and new cardinalities. This is done until all contradictions are solved.

6.2.7 Concept Abduction

After concept contraction it can still be the case that the demand is not able to "call" a supply signature. This is due to missing information in the demand, rather than contradicting information.

The algorithm takes every parameter from the supply and checks whether the demand contains it. If not it is hypothesized to the demand, meaning it is added with the appropriate cardinality to a hypothesis list. There are several critical steps partly due to the way the signature model is designed. The inclusion of a parameter in a signature means also that the (transitive) super-types (even if no parameters directly of this type are present in the signature) are also present. This menas one has to carefully check if the cardinality of the parameter was already included or not. There are three possibilities:

1. The type and all transitive parents are not yet part of the hypothesis list:

   Add it and add corresponding penalty. Also add all transitive parents, with the types cardinality.

2. The type is already part of the hypothesis (because of 1.):

   If the cardinality is greater than the one in the hypothesis list, it means that the type was only added as a (transitive) parent. Thus one needs to change the cardinality accordingly and add a penalty for difference.

3. A transitive parent of the type is already included:

   This means the penalty for the parent already included the penalty from this type. The type is added to the hypothesis but no additional penalty is calculated.

Once the algorithm is finished, it returns a list, which contains the return types of the supply signatures and the corresponding penalties.

Chapter 7. Evaluation Of The Prototype

Summary. *The previous chapter has shown that concept contraction and concept abduction can find fuzzy matches and lead to a ranking of results. This chapter looks at two things: how effective is the algorithm and how efficient is it. The effectiveness is tested through a series of manually conducted studies, while the efficiency is tested through a series of performance tests.*

7.1. Effectivness

The previous chapter has shown a sample implementation of the proposed algorithm. To validate the proof of concept a series of tests have been conducted. The testing procedure consisted of two parts.

First the effectiveness of the algorithm was tested by a comparison of rankings done by human users and by the algorithm. The test users had to rank sample supply signatures according to a given demand signature. This was done for three different demand signatures. The results are compared with the rankings done by the algorithm, based on the three classes of penalty functions discussed before. The full test details can be found in the appendix.

Figures 7.1 through 7.3 show the results of these tests. Each graph shows the ranking done by human users, compared to the ranking achieved by applying all three types of penalty functions. The graphs show three things. First the algorithm works (within some boundaries). Second the results heavily depend on the choice of penalty function – which was expected – and thirdly the most simply penalty function is the most accurate one – which was not expected. Figure 7.4 shows the mean deviation. The penalty functions of class A have in all three test cases the lowest deviation from the human input.

7.2. Efficiency

The second part of tests conducted are related to the performance of the algorithm. This is not so easy, because most of the parts of the algorithm rely

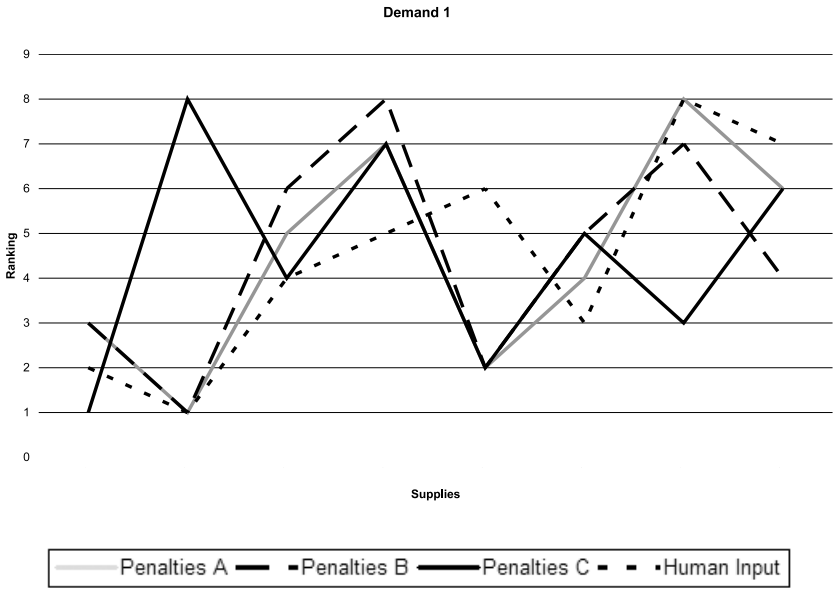


Figure 7.1: Test results for demand 1

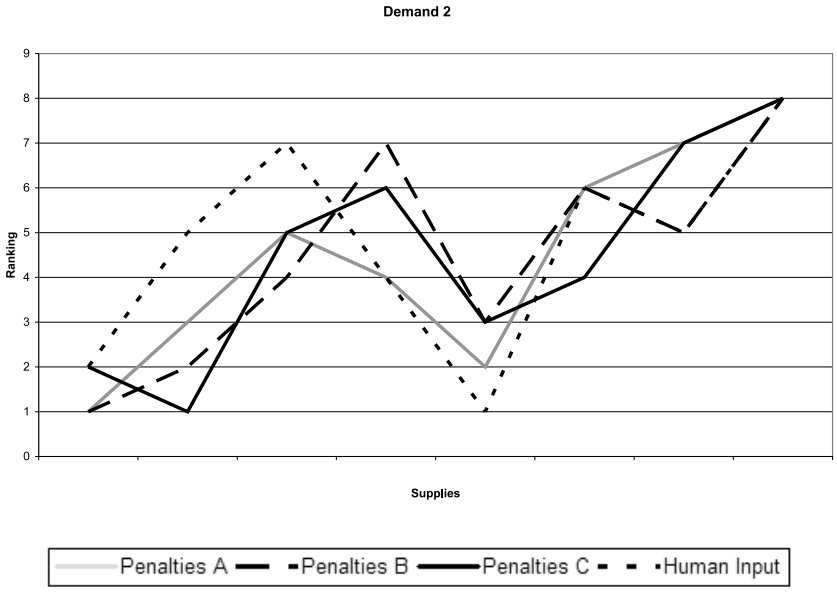


Figure 7.2: Test results for demand 2

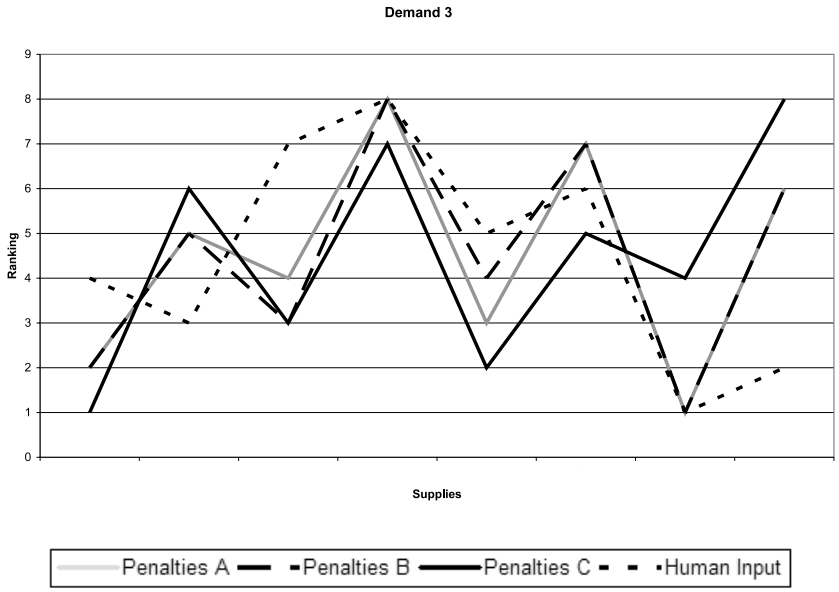


Figure 7.3: Test results for demand 3

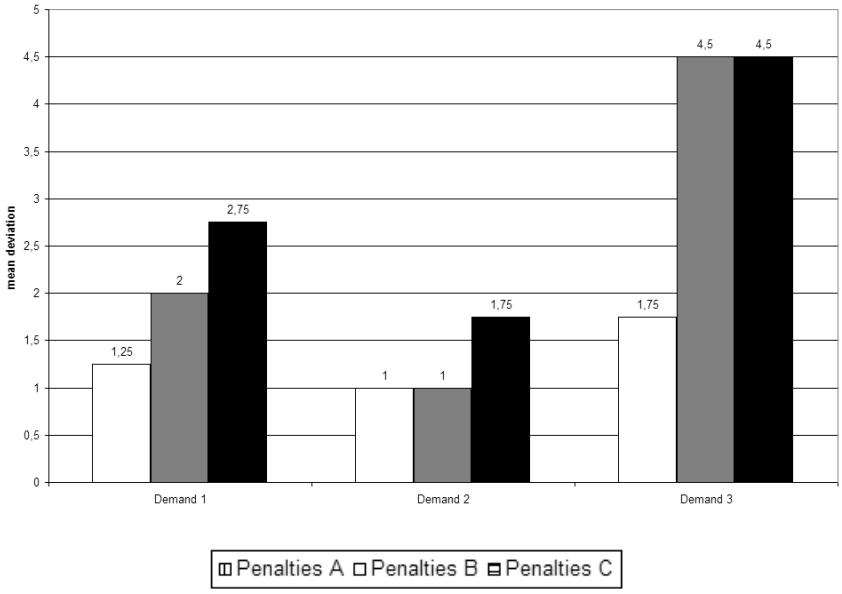


Figure 7.4: Mean deviation of penalty functions from human input

on the employed description logic system. For the example implementation the RacerPro system was used. The interface is based on sockets and text strings that are sent/received to/from the RacerPro system. This in itself is a very costly operation, because every query consists of the following steps:

1. The connection to the RacerPro server has to be openend. This means the socket has to be opened and the input/output streams to be created.

2. The message has to be send to the server. This consists of sending the message via the open socket and parsing the result returned.

3. The connection has to be closed finally, which means the open streams and sockets have to be closed.

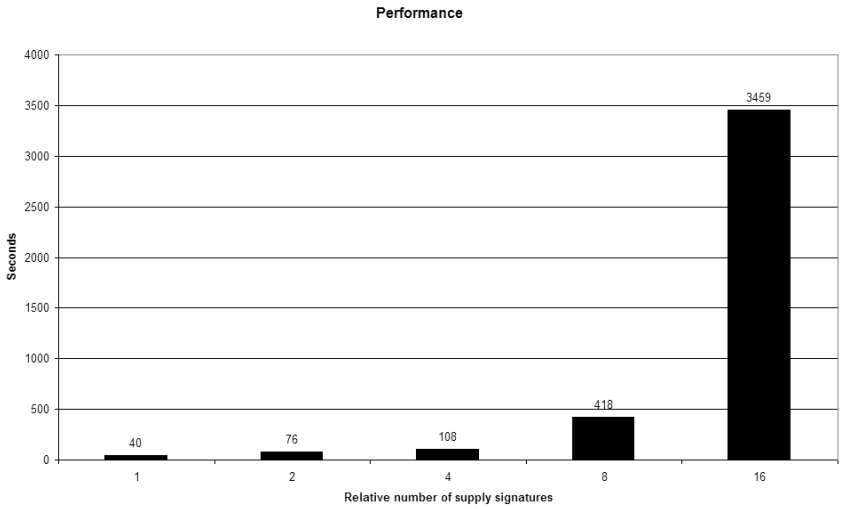


Figure 7.5: Seconds needed for calculation over relative number of supply signatures.

Figure 7.5 shows the results of the test. It shows the relative number of supply signatures in contrast to the seconds taken for the calculation. Although this chart shows a non-linear progression, several factors indicate that this is still the case for the algorithm. Firstly for a demand signature with n parameters and a supply signature with m parameters n + m comparisons have to be done. In addition the RacerPro server is called two times. Once to load the supply signature and once to calculate the subsumption

relationship for these two signatures. For each additional signature in the supply, this cycle is done again, so there is no hint, that the performance scales non-linearly.

A sound explanation for the slow performance is the high memory use of the RacerPro server. The tests were conducted on laptop with a 1.3 Ghz processor and 512 MB RAM. The processor is not used at its full capacity, but memory requirements of the server scale exponentially with the number of supply signatures. This is due to the fact that all inferences that are made are stored in memory. Due to this the runtime of the algorithm is cut by a factor of around 50, if the same queries are done repeatedly, because the inferences are then taken from the cache. Though these are nice results, they don't reflect a normal situation, because usually a query will not have been done beforehand.

Chapter 8. Summary and Conclusion

Summary. *The previous chapters have shown that concept contraction and concept abduction are valid solutions to finding relaxed matches. This chapter gives a conclusive review of the whole thesis. Observations made during the development of the thesis are discussed. The thesis is concluded with an outlook at future research topics.*

8.1. Contributions

Content or knowledge management is an imminent problem of our society. While (almost) everyone has gained access to new and – especially – cheap means to make his knowledge available to others, means to manage this wealth of knowledge have yet to be developed. This thesis has highlighted some of the main areas of concern associated with this and has shown solutions to some of them.

Content classification is one of the starting points when searching for means of structuring arbitrary types of content. This first step was to decide on a system to represent knowledge. It was shown that description logic systems are very well equipped for this task. The approach taken was to develop a signature model that is very general and can be used very flexibly for arbitrary types of content. Signature matching was then discussed as the solution for the search and retrieval of content. The focus was not on finding exact matches, but relaxed matches which take inconsistencies and incompleteness into account. Many different studies are currently done in this field and this thesis relates signature matching to some of the major problems of knowledge management. It was shown that content classification with the help of matching algorithms is a viable method that leads to a useful ranking of alternatives.

In addition, this thesis proved that concept contraction as well as concept abduction can be successfully used to tackle the problems of inconsistent and incomplete information. These concepts are able to find contradictions and missing information in signatures. They are also well equipped to qualify

the inconsistencies found. Based on these advanced inferencing services a signature matching algorithm was developed and tested in a specific application domain. It was shown that it compares well to human input if the right penalty functions are used.

8.2. Observations

The implemented prototype has shown that the proposed algorithm works. Signature matching by concept contraction and concept abduction helps to rank relaxed matching results and gain sufficiently adequate results. It was shown that the results of the prototype are very close to the decisions made by human users, but that the results depend heavily on the used penalty function. It is necessary to highlight, that the simplest penalty function was the most accurate.

The prototype has also shown that the algorithm depends heavily on the used description logic system for its inferencing system. The algorithm in itself works very efficiently and especially a clever implementation of the object representation of the signatures has proved to be very useful. The dependence on the description logic system makes it necessary tough to conduct further research with different description logic systems.

The biggest surprise was the difficulty in finding correct values for the parameters of the penalty functions. Seemingly endless possibilities regarding constants and weights make extensive testing and evaluation necessary. The tested values show that the results depend heavily on the used penalty algorithms and the applied weights.

8.3. Future Directions

The proposed signature model is quite flexible, as it allows users to develop mappings from arbitrary descriptions schemes. Asset Expressions are one example, but as long as some piece of content is described via a constant set of types or attributes, a mapping can be created. The signature model though has the problem, that it only captures a very small amount of the information made available through descriptions. Only parameter types and their cardinality are examined. Several other types of information could be incorporated into the signature model. Examples would be a differentiation of different type classes, similar to the differentiation of strict and negotiable

parameters done in [Colucci et al., 2004b]. It would also be possible to assign a weight to each parameter indicating its importance for the understanding of the piece of content.

A second interesting point is the application of the algorithm to areas other than classification. The proof of concept has shown that concept contraction and concept abduction can successfully and efficiently be employed to classify medial content. The first step would be to think of different types of content, that could benefit from such a characterization. One idea is the development of directory services. In our days users employ search engines to manage the data on their home computers. Building a signature with hypotheses about the types of documents the user wants to find, would be an interesting topic to follow. Other applications are the classification, search and retrieval of web-services or the development of web search engines based on contextual knowledge.

The proof of concept implementation has shown that the design of effective penalty functions is crucial for reliable and credible results. The discussed classes of penalty functions have proved to be a sensible start, but even in these (relatively restricted) examples small changes to the weights can have severe impact on the results. Two paths for improvement can be clearly derived from the experiences made in this thesis. Firstly it has to be investigated if contraction and abduction are the only steps in the algorithm that need to be penalized. As exact matches are found through generalization (employing the subsumption relationship of the description logic system), it might be necessary to add penalties to generalization. Although additional information does not interfere with the classification process, it still says something about the content under consideration. One could argue for example, that the more specialized (meaning the more additional information is present) a signature is, the farther away it is from a general signature subsuming it (this would only hold true, if the modeling of the application domain is equally detailed on every level of the subsumption hierarchy).

A last point are improvements to the general performance of the algorithm. The sample implementation has shown that the biggest bottleneck is the communication with the description logic system. Right now the Java socket implementation severely hampers the performance. Thus it would be interesting to investigate alternatives. The obvious one is to integrate

contraction and abduction into the description logic system. The advantage is evident, but the problem lies in the "special" requirements of the signature model. As is it does not use standard contraction and abduction, but a specialized version, this might require a special implementation, too. The second problem with this approach is that it would still be necessary for the signature matching algorithm to call these services in the description logic system, meaning communication with the description logic system would increase. A different approach to this problem would be to completely integrate the necessary description logic function into the signature matcher and use the description logic system only as a kind of advanced data storage.

Bibliography

Geschichte der Kunstgeschichte des Nationalsozialismus (GKNS). www.welib.de/gkns/index.html.

Racerpro. www.racer-systems.com.

Warburg electronic library. www.welib.de.

F. Baader, D. Calvanese, G. D. Giacomo, P. Fillottrani, E. Franconi, B. C. Grau, I. Horrocks, A. Kaplunova, D. Lembo, M. Lenzerini, C. Lutz, R. Moeller, B. Parsia, P. Patel-Schneider, R. Rosati, B. Suntisrivaraporn, and S. Tessaris. Formalisms for representing ontologies: State of the art survey. May 2005.

F. Baader and R. Küsters. Computing the least common subsumer and the most specific concept in the presence of cyclic $\mathcal{ALN}$-concept descriptions. In O. Herzog and A. Günter, editors, *Proceedings of the 22nd Annual German Conference on Artificial Intelligence, KI-98*, volume 1504, pages 129–140, Bremen, Germany, 1998. Springer–Verlag. URL `citeseer.ist.psu.edu/baader98computing.html`.

F. Baader and R. Küsters. Matching in description logics with existential restrictions. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the Seventh International Conference on Knowledge Representation and Reasoning (KR2000)*, pages 261–272, San Francisco, CA, 2000. Morgan Kaufmann Publishers. URL `citeseer.ist.psu.edu/baader99matching.html`.

A. Borgida and R. J. Brachman. *The description logic handbook: theory, implementation, and applications*, chapter Conceptual modeling with description logics, pages 349–372. Cambridge University Press, New York, NY, USA, 2003. ISBN 0-521-78176-0.

J. Börstler. Feature-oriented classification for software reuse. *SEKE*, pages 204–211, 1995.

S. Boßung. *Conceptual Content Modeling*. PhD thesis, Technical University Hamburg Harburg, 2007.

S. Brandt, R. Kusters, and A. Turhan. Approximation and difference in description logics. In *The 8th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'2002)*, pages 203–214, 2002. URL `citeseer.ist.psu.edu/brandt02approximation.html`.

S. Brandt and A.-Y. Turhan. Using non-standard inferences in description logics — what does it buy me? In *Proceedings of the KI-2001 Workshop on Applications of Description Logics (KIDLWS'01)*, number 44, Vienna, Austria, 2001. RWTH Aachen. URL `citeseer.ist.psu.edu/467528.html`.

A. Calì, D. Calvanese, S. Colucci, T. D. Noia, and F. M. Donini. A description logic based approach for matching user profiles. In *Description Logics*, 2004.

S. Colucci, T. Di Noia, E. Di Sciascio, F. M. Donini, and M. Mongiello. Concept abduction and contraction in description logics. In *Proceedings of the 16th International Workshop on Description Logics (DL'03)*, volume 81 of *CEUR Workshop Proceedings*, Sept. 2003. URL `http://sisinflab.poliba.it/sisinflab/publications/2003/CDDDM03a`.

S. Colucci, T. D. Noia, E. D. Sciascio, F. M. Donini, and M. Mongiello. A uniform tableaux-based approach to concept abduction and contraction in aln. In *Description Logics*, 2004a.

S. Colucci, T. D. Noia, E. D. Sciascio, M. Mongiello, and F. M. Donini. Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, pages 41–50, New York, NY, USA, 2004b. ACM Press. ISBN 1-58113-930-6. doi: http://doi.acm.org/10.1145/1052220.1052226.

T. Dean, editor. *Computing Least Common Subsumers in Description Logics with Existential Restrictions*, 1999. Morgan Kaufmann. URL `citeseer.ist.psu.edu/article/baader98computing.html`.

T. Di Noia, E. Di Sciascio, F. M. Donini, and M. Mongiello. Abductive matchmaking using description logics. In *Proceedings*

of Eighteenth International Joint Conference on Artificial Intelligence IJCAI-03, pages 337–342, Acapulco, Mexico, Aug. 2003. MK. URL `http://sisinflab.poliba.it/sisinflab/publications/2003/DDDM03`.

I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic SHIQ. In D. MacAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, number 1831, Germany, 2000. Springer Verlag. URL `citeseer.ist.psu.edu/article/horrocks00reasoning.html`.

M. Klein and B. König-Ries. Coupled signature and specification matching for automated service binding. In *Lecture Notes in Computer Science*, volume 3250, 2004.

R. Loader. Notes on simply typed lambda calculus, May 2006. URL `http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/`.

T. Mantay. Computing least common subsumers in expressive description logics. In *Australian Joint Conference on Artificial Intelligence*, pages 218–230, 1999.

D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook Theory, Implementation and Applications*. Cambridge University Press, 2003.

R. Möller, V. Haarslev, and B. Neumann. Semantics-based information retrieval. In *Proc. IT&KNOWS-98: International Conference on Information Technology and Knowledge Systems, 31. August- 4. September, Vienna, Budapest*, pages 49–6, 1998.

T. D. Noia, E. D. Sciascio, F. Donini, and M. Mongiello. A system for principled matchmaking in an electronic marketplace. In *The Twelfth Int'l Conf. on World Wide Web (WWW'03)*, pages 321–330, Budapest, Hungary, May 2003. URL `citeseer.ist.psu.edu/dinoia03system.html`.

P. Rosenbloom and P. Szolovits, editors. *Computing Least Common Subsumers in Description Logics*, Menlo Park, California, 1992. AAAI Press. URL `citeseer.ist.psu.edu/cohen92computing.html`.

G. Teege. Making the difference: A subtraction operation for description logics. In *KR'94: Principles of Knowledge Representation and Reasoning*, pages 540–550, San Francisco, California, 1994. Morgan Kaufmann. URL `citeseer.ist.psu.edu/teege94making.html`.

W3C. Owl web ontology language overview. http://www.w3.org/TR/owl-features/.

H. S. Wilf. *Algorithms and Complexity*. A.K.Peters Ltd., 1994.

A. M. Zaremski. *Signature and Specification Matching*. PhD thesis, 1996. URL `citeseer.ist.psu.edu/article/zaremski96signature.html`.

A. M. Zaremski and J. M. Wing. Signature matching: A key to reuse. In D. Notkin, editor, *Proceedings of theFirstACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 182–190. ACM Press, 1993. URL `citeseer.ist.psu.edu/zaremski93signature.html`.

A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. In *ACM Transactions on Software Engineering and Methodology*, volume 4, pages 146–170, 1995.

Appendix A. The Effectiveness Test

A.1. General information

To test the effectiveness of the algorithm a test suite was developed. It consists of a base library of eight supply signatures. These supply signatures are all classified according to a specific type hierarchy – the food type hierarchy (which can be found on the attached data CD, see appendix B). The signatures were presented to the users as pictures annotated with asses expressions. Thus the structure of each signature picture is as follows: In the middle the classified content can be found, a picture of a specific meal. This piece of content is annotated with abstractions. Where possible the corresponding application to each abstraction (meaning a picture of the food type used) were also added. The test users were given this library of signature pictures and in addition three demand signature. These demand signatures are pictured exactly like the supply signature, with the exception being the missing classification of the central content. The task every user then had to fulfill was to create a ranking of the eight supply signatures according to how close they are to the demand signature.

Altogether 16 test users created sample rankings. The mean of these rankings was then compared with several different instances of the developed signature matching algorithm. The results are shown in chapter 7

A.2. The Sample Demand And Library Signatures

On the following pages the signature pictures, that were presented to the test users are shown.

Figure A.1: Sample library signature 1

Figure A.2: Sample library signature 2

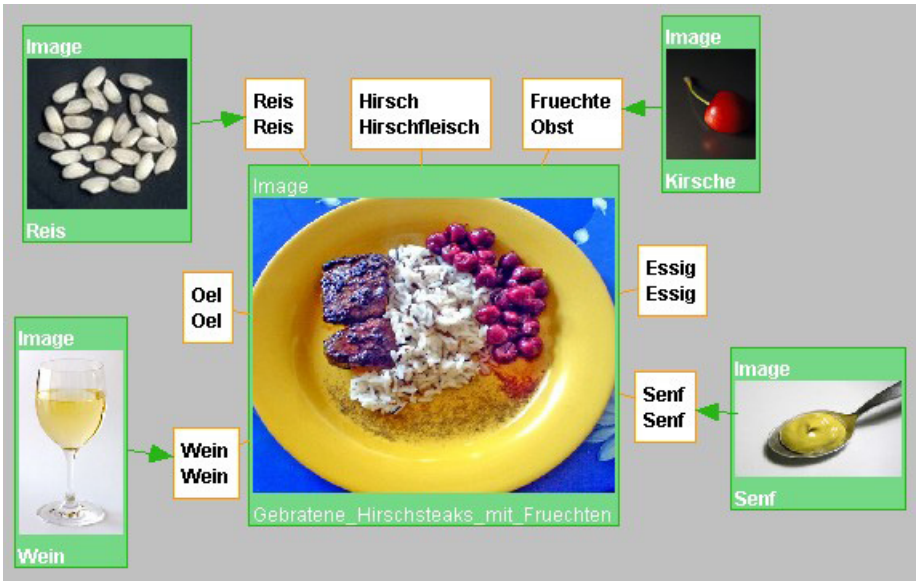


Figure A.3: Sample library signature 3

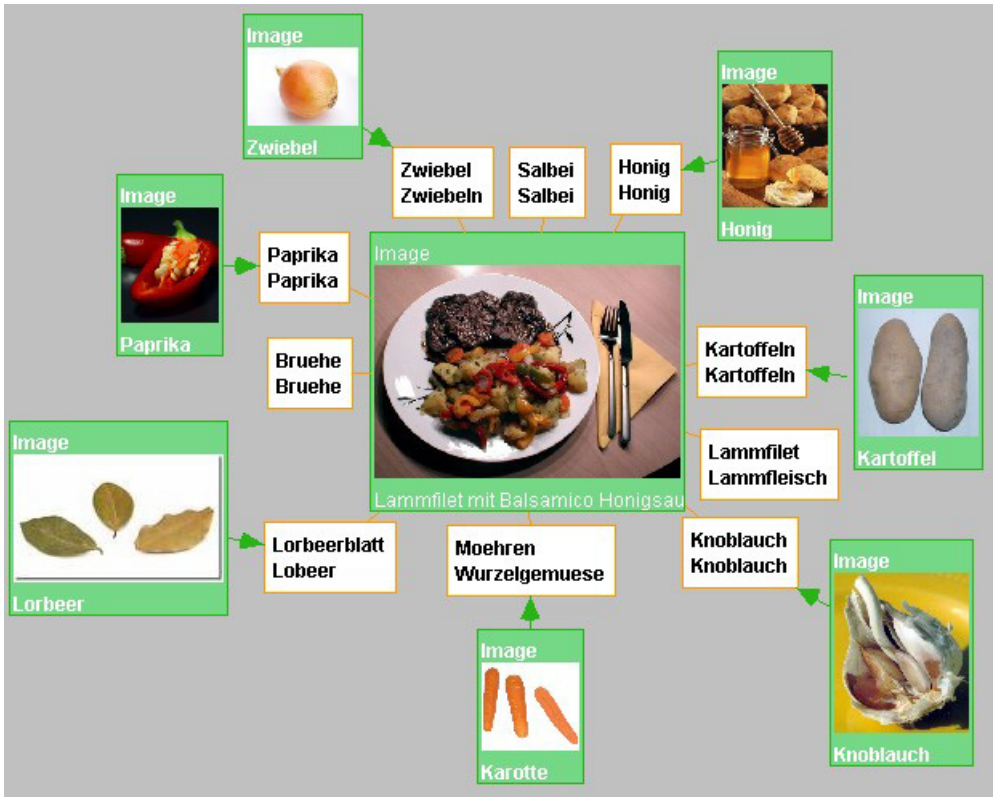


Figure A.4: Sample library signature 4

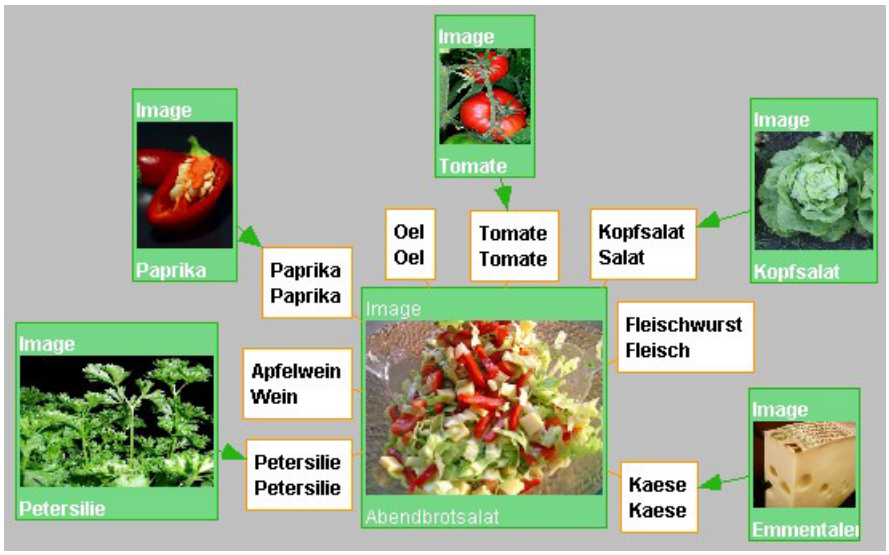


Figure A.5: Sample library signature 5

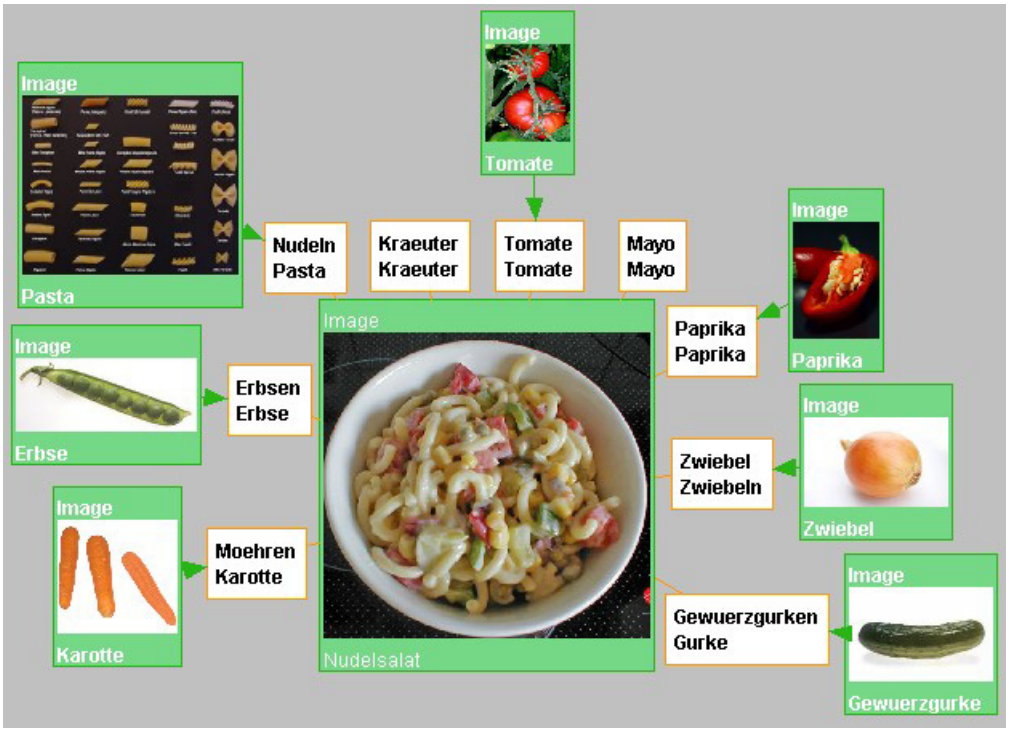


Figure A.6: Sample library signature 6

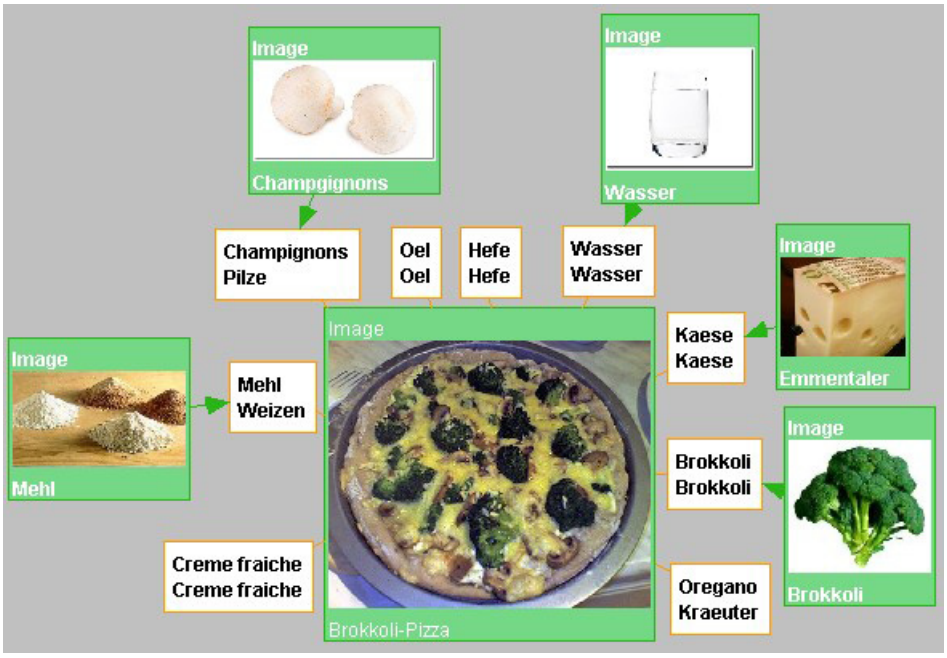


Figure A.7: Sample library signature 7

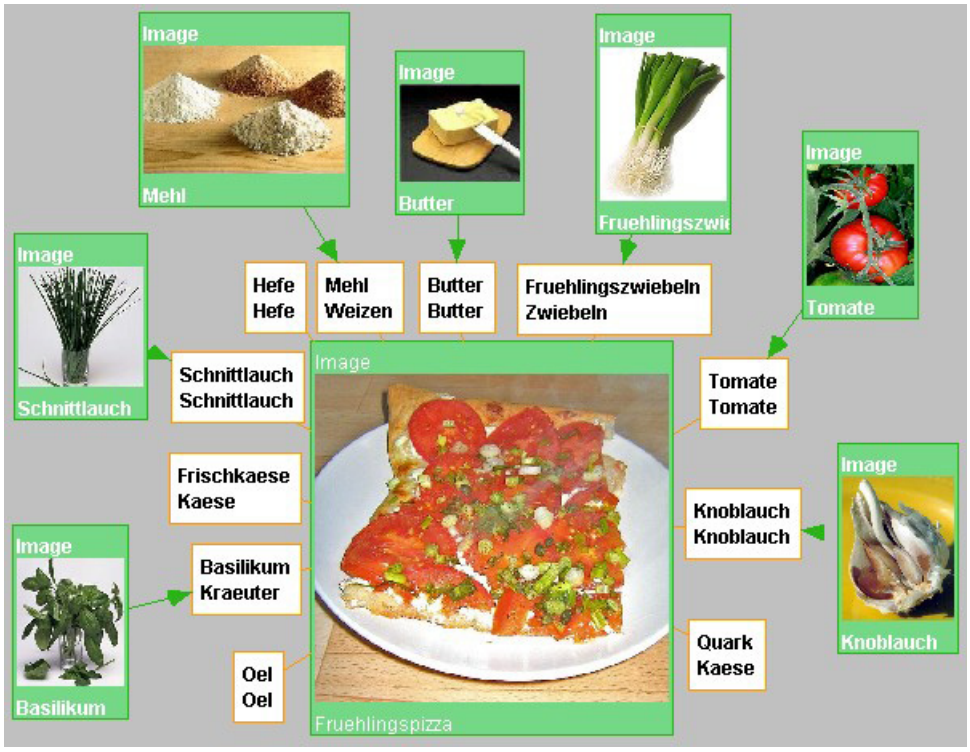


Figure A.8: Sample library signature 8

Figure A.9: Sample demand signature 1

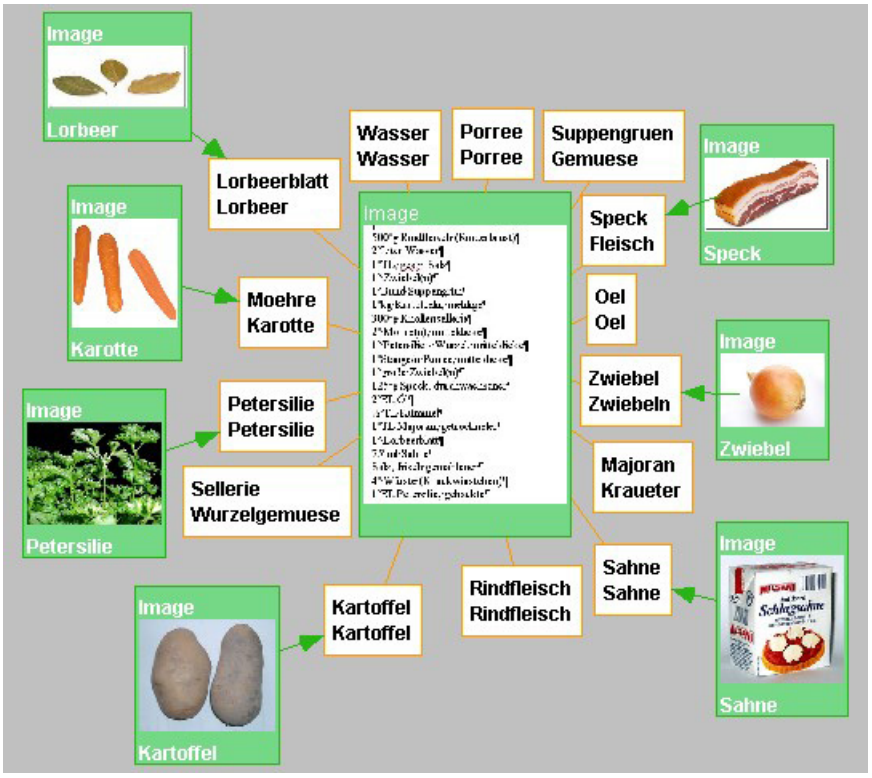


Figure A.10: Sample demand signature 2

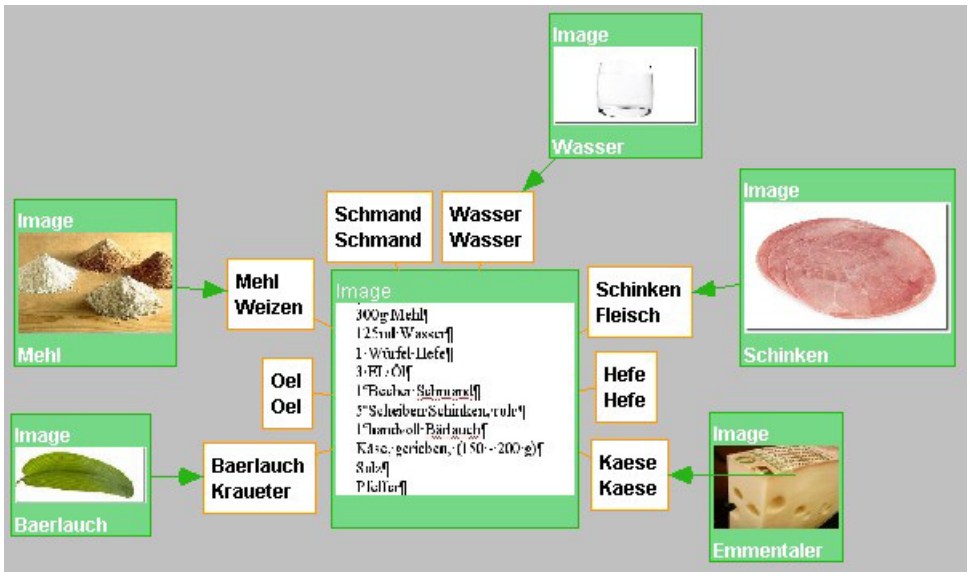


Figure A.11: Sample demand signature 3

Appendix B. The CD

Attached you will find a data CD that contains files related to this thesis. The folders on the CD are structured as follows:

 **Literature** - The references literature in PDF format, as far as it is available publically.

 **Sourcecode** - The whole sourcecode of the developed signature matching algorithm.

 **Thesis** - This thesis in PDF and Postscript format.