National University of Ireland, Galway
Ollscoil na hÉireann, Gaillimh

Diploma Thesis

Simplifying Semantic Annotations in Text

cand. inform. Sebastian Gerke

submitted on June, 1st 2007

Supervisor:
Prof. Dr. Stefan Decker and Eyal Oren
Digital Enterprise Research Institute
National University of Ireland, Galway

2nd supervisor:
Prof. Dr. Rudi Studer
Institut für Angewandte Informatik und Formale Beschreibungsverfahren
Universität Karlsruhe

To my father.

Acknowledgements

I would like to thank Eyal Oren for his great support during the internship and the creation of this thesis. I also thank Prof. Dr. Stefan Decker who gave me the opportunity to work in such a pleasant environment. Furthermore I would like to thank Prof. Dr. Rudi Studer and Max Völkel for arranging the internship and for co-supervising this thesis; all people in DERI and my flat mates for making my stay in Galway an unforgettable one.

I want to say thank you to my mother and my late father for making my studies possible. They taught me things that really matter in life. And last but not least, I would like to thank Judith for her love and support throughout my studies.

Contents

Zusammenfassung

Die Vorteile semantischer Annotationen sind vielschichtig: Sie erlauben detailliertere Abfragen als konventionelle Suchmaschinen, durch sie können Fragen präzise beantwortet werden (anstelle des Auffindens mehr oder wenig relevanter Dokumente) und sie ermöglichen regelbasierte Schlussfolgerungen. Kurz: Sie erleichtern die Arbeit. Allerdings erfordert die Erstellung semantischer Metadaten zusätzlichen Aufwand, der den Gesamtvorteil semantischer Technologien verringert. Die vorliegende Arbeit stellt verschiedene Ansätze vor, die die Erstellung semantischer Annotationen in Texten vereinfachen. Der Fokus ist hierbei auf semantische Wikis als Autorenumgebung gelegt.

Folgende Probleme im Umfeld der Erstellung semantischer Annotationen werden in dieser Arbeit behandelt: Zuerst wird ein Wiki-Parser vorgestellt, der es erlaubt, die Inhalte eines Wikis syntax-unabhängig zu speichern. Dadurch können die Benutzer eines Wikis ihre bevorzugte Wiki Syntax benutzen, sie müssen nicht eine festgelegte Syntax benutzen. Um die Inhalte eines Wikis abzuspeichern, wird ein Wiki-Datenmodell eingeführt.

Existierende Daten auf dem Desktop eines Benutzers sind oft auf "Dateninseln" verteilt: Veschiedene Programme speichern ihre Daten zwar in maschinenlesbarer Form ab, stellen sie jedoch nicht anderen Anwendungen zur Verfügung. So ist es nur schwer möglich, Verknüpfungen zwischen Ressourcen aus verschiedenen Anwendungen zu erstellen. In dieser Arbeit wird eine Architektur zur Integration existierender Daten in RDF-basierte Programme vorgestellt. Dazu wird *ActiveRDF*, eine Datenspeicher-unabhängige Bibliothek zum objektorientierten Zugriff auf RDF-Daten, erweitert, um Daten verschiedener Programme wie RDF-Daten zu behandeln. Das ermöglicht es Programmierern, Daten aus beliebigen Anwendungen und RDF-Datenquellen zu kombinieren.

Ein weiteres Problem ist die Verwendung eines gemeinsamen Vokabulars, um Bedeutung aus den semantischen Annotationen zu generieren. Erst ein gemeinsames Vokabular macht das Semantic Web semantisch. Im Gegensatz zum Tagging, wie in vielen sog. "Web 2.0" Anwendungen gebräuchlich, ist die Verwendung festgelegter Annotationen notwendig. Diese Annotationen müssen zum Teil in Spezifikationen nachgeschlagen werden, da sie dem Benutzer häufig nicht in vollem Umfang bekannt sind. Um das Nachschlagen zu vermeiden beziehungsweise zu vereinfachen, werden zwei Systeme vorgestellt, die Annotationen für Ressourcen vorschlagen. Das erste System extrahiert Informationen aus natürlichsprachlichem Text, um Vorschläge zu generieren, während das zweite System bereits bestehende Annotationen als Grundlage für weitere Annotationen verwendet.

Das natürlichsprachliche System basiert auf der Übereinstimmung von Schlagworten zwischen dem Text und den lokalen Namen von URIs in RDF Schemata. Dieser einfache Ansatz dient als Basis für komplexere Algorithmen und verdeutlicht die Integration eines solchen Systems in den Autorenprozess.

Für den zweiten, auf existierenden Annotationen basierenden Ansatz werden zwei Algorithmen vorgestellt. Der erste Algorithmus verwendet einen Klassifikator, um ähnliche Ressourcen zu identifizieren und schlägt anschließend Annotationen vor, die häufig von ähnlichen Ressourcen verwendet werden. Verschiedene Variationen dieses Algorithmus' wurden getestet, welche sich unter anderem in der Definition eines Ähnlichkeitsmaßes unterscheiden. Die qualitative Evaluation ergab vielversprechende Resultate ($F_1 \geq 0,85$), während die Laufzeit des Algorithmus' nicht zufriedenstellend skalierte, da sie linear abhängig von der Anzahl der Ressourcen in der Wissensbasis ist. Die Verwendung von Untermengen der gesamten Wissensbasis ergab keine zufriedenstellende Verbesserung der Laufzeit ($> 2s$) ohne die Qualität des Algorithmus entscheidend zu verschlechtern. Der zweite Algorithmus verwendet gemeinsame Auftretenshäufigkeiten zwischen Annotationen. Dadurch können einige Berechnungen im Voraus vorgenommen werden, was in einer stark beschleunigten Erstellung von Vorschlägen resultiert. Darüber hinaus ist die Laufzeit nicht mehr von der Anzahl der Ressourcen abhängig, sondern nur noch von der Anzahl der verwendeten Prädikate in der Wissensbasis. Das führt dazu, dass eine größere Wissensbasis nicht notwendigerweise die Laufzeit verschlechtert. Evaluationen haben gezeigt, dass Vorschläge in ca. 0,01 Sekunden generiert werden können und deren Qualität geringfügig besser ist als die des Klassifikator-Ansatzes.

Abstract

The advantages of Semantic annotations are extensive: They allow more detailed queries than conventional search engines, precise question-answering instead of returning more or less relevant resources, and rule-based reasoning. Briefly: They can save work. But the creation of semantic metadata requires additional effort that decreases the overall benefit of semantic technologies. The advantage of semantic metadata should outweigh its additional creation cost. This thesis proposes different techniques for simplifying the creation of semantic annotations in text, with a focus on semantic wikis as a semantic authoring environment.

Different problems of semantic authoring are tackled in this thesis: First, a wiki parser which allows a syntax-independent storage of content is presented. It allows users to use their preferred wiki syntax, they don't have to adopt to a fixed syntax. A wiki data model is introduced to store the wiki content.

Existing data on the desktop is often spread into "data islands": different applications already store their data machine-readable, but usually it is not directly available to other applications. It is not possible to create links between resources from different applications. In this thesis, an architecture that facilitates integration of existing data into one semantic application is presented. It extends *ActiveRDF*, a data store independent RDF library that allows object-oriented access to RDF data, to provide access to this desktop data as if it were RDF data. That lets programmers seamlessly combine data from RDF data stores and desktop applications in their programs.

Another problem in semantic knowledge acquisition and authoring is that users have to use a shared vocabulary to create meaning out of annotations. This is a tedious task because finding the appropriate vocabulary often implies looking up vocabulary specifications to ensure a correct usage of vocabulary. To simplify vocabulary lookup, two systems for suggesting annotations for a resource are presented in this thesis. The first suggestion system is based on natural language text, whilst the second is based on existing annotations.

The natural language based system applies keyword matching in the text against keywords and local names of URIs in RDF schemas. Due to this simple approach, the results of this system are not optimal. It can serve as a foundation for more sophisticated algorithm for annotation suggestion based on natural language text.

For the second approach, where the suggestions are based on existing annotations, two different algorithms are proposed. One is using a classifier approach to identify similar resources and then suggest annotations that are often used among these similar resources. Similar resources are determined

by a similarity measure. Different variations of the algorithm were tested. The qualitative results are quite good ($F_1 \geq 0.85$), but the runtime performance does not scale well, it is linearly dependent from the number of resources in the knowledge base. Using only a subset of all resources does not yield satisfactory runtime performance ($> 2s$) without sacrificing too much qualitative performance. The second algorithm for annotation suggestion based on existing annotations uses co-occurrences of predicates. A lookup table containing co-occurrences of all predicates can be computed beforehand. At query time, only this table is used to generate suggestions. The query runtime performance then only depends on the number of different predicates used, not on the number of resources. That means that a bigger knowledge base does not necessarily results in a worse runtime performance. Evaluations of the algorithm show that suggestions are generated in about 0.01 seconds, yielding a F_1 measure that is slightly better than those of the similarity-based algorithm ($F_1 = 0.87$).

Chapter 1

Introduction

The Semantic Web extends the current web with machine-readable metadata. The *World Wide Web Consortium* (W3C), the main standardisation organisation for the World Wide Web, describes the Semantic Web as follows [17]:

> "The Semantic Web is a web of data. There is lots of data we all use every day, and its not part of the web. I can see my bank statements on the web, and my photographs, and I can see my appointments in a calendar. But can I see my photos in a calendar to see what I was doing when I took them? Can I see bank statement lines in a calendar?
>
> Why not? Because we don't have a web of data. Because data is controlled by applications, and each application keeps it to itself.
>
> The Semantic Web is about two things. It is about common formats for integration and combination of data drawn from diverse sources, where on the original Web mainly concentrated on the interchange of documents. It is also about language for recording how the data relates to real world objects. That allows a person, or a machine, to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing."

But making the World Wide Web a web of data is expensive, especially when it is not created automatically. Human-generated content gains more and more popularity in the so-called Web 2.0. Blogs, wikis and other community-centric web sites are emerging rapidly during the last few years. Making the content on these web sites machine-processable is laborious, additional effort is required to enrich the data semantically. A user creating content on a wiki e.g. also has to enter additional semantic annotations about the things he describes on a wiki page. To increase the overall benefit from semantically annotated web content, one can either improve the usage

of semantically annotated content, or one can facilitate creation of semantic metadata. In this thesis, we focus on the latter, i.e. we introduce methods for simplifying the creation of semantic annotations in text, especially in wiki text.

The thesis is organised as follows: First, we explain the background technologies of this work in chapter 2. That includes an introduction to Semantic Web and its technologies, a short overview over semantic wikis and a brief summary of ActiveRDF, a library for the Ruby programming language that provides an object-oriented access to Semantic Web data. Afterwards, the developed methods are described in detail. In chapter 3, we present a method for a personalisable wiki syntax. In chapter 4, we present an architecture and a prototype implementation of a desktop integration component which allows for reuse of existing data and metadata on the user's desktop. In chapter 5, we show a prototype of a annotation suggestion system that uses natural language from text to generate suggestions. In chapter 6, we introduce a annotation suggestion system that gives suggestions for further annotations based on existing ones. Finally, we conclude the presented work in chapter 7. Additional tables and plots that show additional results from chapter 6 can be found in the appendix. They were moved to the appendix due to readability reasons.

Chapter 2

Background

2.1 Semantic Web and the role of RDF

The Semantic Web is a machine-readable extension of the current World Wide Web. In its current form, data on the Web is mainly supposed to be consumed by humans. Machines can not understand the *meaning* of the majority of content on the Web and thus can not use this information. So if a user wants to aggregate or combine data from different sources, he has to do it manually.

Imagine the scenario of preparing a trip to a conference. The user first has to search for a flight to the conference venue, then book this flight. After that, he has to check the timetable of the local train company to get a train to the airport. He reserves a seat over the online reservation system of the train company. Finally he searches for a bus service that operates from the airport to the city center and checks the timetable for a bus connection.

In an ideal Semantic Web, all pages on the Web are semantically annotated, i.e. each page provides machine-readable metadata that describes its content. The current content is not replaced by machine-readable data, it is enriched by it. Originally, Tim Berners Lee expressed this vision as follows [6]:

> "I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers.[...] the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize."

In a Semantic Web setting, the whole trip planning process would be like this: The users tells a software agent that he wants to be at a specified hotel at a specified time. The agent then automatically checks all necessary timetables (train, flight and bus) and presents a list of possible trips along

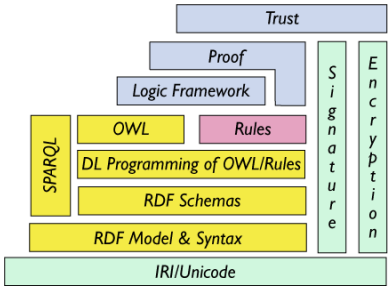


Figure 2.1: Semantic Web Stack

with their prices and itineraries. The user then has to select one trip and the agent immediately books all tickets and prints them.

The Semantic Web uses ontologies as a formal representation of data. An ontology is a shared data model that represents a domain and is used to reason about the objects in that domain and their relations. An ontology defines classes, instances of classes, attributes and relations between instances (like a taxonomy). Furthermore, it provides rules for reasoning over the knowledge encoded in this ontology. Popular ontologies are the Friend Of a Friend (FOAF), Dublin Core (DC) and Socially Interlinked Online Communities (SIOC) ontologies.

Figure 2.1 shows the so-called Semantic Web layer cake that depicts the standard technologies that are used in Semantic Web applications. The bottom layers consist of existing web standards for identifying resources (uniform resource identifier (URI)) and character encoding (Unicode)[1]. URIs will be explained more detailed in section 2.1.1. XML, a standard syntax for data, uses these two technologies to build a foundation for the Semantic Web specific technologies.

The first Semantic Web specific layer is the Resource Description Framework (RDF). RDF is a "data model for referring to resources and how they are related"[39]. It provides a standard data model for encoding knowledge in triples of elementary sentences containing a subject, a predicate and an object. A more detailed explanation of RDF can be found in section 2.1.2. On top of RDF is RDF Schema (RDFS). It is a lightweight ontology language for specifying vocabularies for RDF [12]. RDFS is an extension of RDF which provides a vocabulary for defining classes, class hierarchies, properties, property hierarchies and property restrictions [23]. This vocabulary allows basic reasoning over subclass and subproperty hierarchies. RDFS is presented more detailed in section 2.1.3. The OWL (Web Ontology Language) / Rules layer contains a vocabulary that allows more advanced resource descriptions and restrictions. OWL consists of three sub-

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

languages: OWL Lite, OWL DL [2] and OWL Full. These three sublanguages differ in expressiveness and computational complexity. Each of the sublanguages extends its simpler predecessor, i.e. a valid OWL Lite conclusion is also a valid OWL DL and OWL Full conclusion [25].

SPARQL (SPARQL Protocol and RDF Query Language) is a protocol and a query language similar to SQL that allows to query RDF stores. Logic is provided by reasoners that allow inferencing of new knowledge based on rules, whereas on the proof layer the consistency of existing data based on the same rules is guaranteed. Finally, the trustfulness of the data can be assured by using signatures and encryption.

2.1.1 URI

An URI (Uniform Resource Identifier) is defined as an extensible, compact string of characters that is used for the identification of resources. A URI can either identify a physical or an abstract resource, like web sites, documents, email addresses, web service endpoints etc. An example for an URI is `http://www.deri.ie/about`. Generally, an URI consists of two parts: the first part (before the colon) defines the URI scheme, the second part is type-specific. Its interpretation depends on the URI scheme.

2.1.2 RDF

The metadata has to be put in a standardized format to ensure exchangeability. The standard format (as specified by the World Wide Web Consortium (W3C)) to express the meaning is the Resource Description Framework (RDF). RDF encodes knowledge in sets of triples (also called statements), each triple representing the subject, predicate and object of an elementary sentence [37]. For example to express the sentence "Eric knows Steve" in RDF, a triple with a subject denoting "Eric", a predicate denoting "knows" and an object denoting "Steve" can be formed.

The subject of a triple is a resource, possibly identified by an URI. Some resources do not have an URI, they are anonymous and called blank nodes [23]. As they are anonymous, they cannot be directly referenced. The predicate is also a resource representing a relationship, but it must not be anonymous. The object is either a (possibly anonymous) resource or a string literal.

Definition 1 (RDF Triple) *Let T be a finite set of triples. T contains the finite sets of URIs U, of local blank node identifiers B and of literals L.*

An RDF triple $t \in T$ is defined as a 3-tuple (s, p, o) with $s \in U \cup B, p \in U$ and $o \in U \cup B \cup L$. The projections $subj : t \rightarrow s \in U \cup B$, $pred : t \rightarrow p \in U$

[2] DL stands for Description Logic

and $obj : t \rightarrow o \in U \cup B \cup L$ *return the subject, predicate and object of a triple respectively.*

In the example, Eric is identified by the URI `http://eric.com/foaf.rdf` and Steve by the URI `http://steve.com/foaf.rdf`. Now everyone can make statements about Eric and Steve by referring to their URI. The relation "knows" has the URI `http://xmlns.com/foaf/1.0/knows`. Now the example statement can be expressed in RDF by a triple containing `http://eric.com/foaf.rdf` as subject, `http://xmlns.com/foaf/1.0/knows` as predicate and `http://steve.com/foaf.rdf` as object.

A set of triples is called a RDF graph [20]. A RDF graph can also be graphically represented as a graph with subject and objects of an statement represented by nodes and predicates represented by labelled directed arcs. Formally, a RDF graph is not a graph in a classical mathematical sense because a predicate (arc) in one statement can appear as a subject or an object (node) in another statement. In most graphical representations of RDF graphs, the resources that appear as subject or objects and predicates are duplicated and appear twice in the figure [15]. Another possibility is to allow edges that connect to other edges instead of nodes. Both options have their advantages, see [15] for further details. A more formally correct way to represent a RDF graph is a hypergraph that treats subject, predicate and objects of a statement as nodes and connects them with one directed "arc". The first approach for RDF graph visualisation is used in this thesis. It is more readable and a formally correct graph representation is not needed for our purposes. The formal definition of a RDF graph is as follows:

Definition 2 (RDF graph) *An RDF graph* G *is a set of triples* T *and is defined as* $G = (V, E, l_V, l_E)$ *where* $V := \{v_x | x \in subj(T) \cup obj(T)\}$ *is a finite set of vertices (subjects and objects) with the labelling function* $l_V : v_x \rightarrow x$ *and* $E := \{e_x | x \in pred(T)\}$ *is a finite set of edges (predicates) with the labelling function* $l_E : e_x \rightarrow x$*. The projections* $source : E \rightarrow V$ *and* $target : E \rightarrow V$ *return the source and target nodes of edges respectively.*

In a graphical representation of an RDF graph, URIs and blank nodes are drawn with ellipses while literals are drawn with rectangles. The shapes representing URIs and literals contain a label with their URI or string literal respectively. Blank nodes are usually drawn without any label. Sometimes their local blank node identifier is used as a label. A directed arc from the subject to the object of a triple connects these two resources. The arc is labelled with the URI of the statements' URI. Figure 2.2 shows two example visualisation of a RDF graph, one as it is used in this thesis (left side) and one hypergraph visualisation.

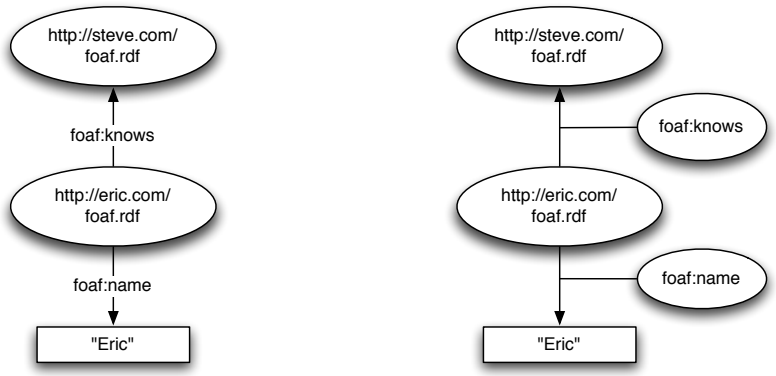


Figure 2.2: Usual graphical representation of a RDF graph (left) and hypergraph representation (right) of the same RDF graph.

Vocabulary

RDF defines a minimal vocabulary to express properties, class memberships, different containers and reification of statements. RDF classes (classes are defined in the RDF Schema vocabulary which is presented in section 2.1.3) and properties are preceded by the prefix `rdf`. `rdf:Property` is the class of RDF properties. Predicates of RDF triples can be inferred as instances of `rdf:Property`. Instances of classes are defined by the `rdf:type` property. A resource can have multiple types, unlike in many object-oriented programming languages.

Reification allows statements about statements by making them adressable. Therefore, the `rdf:statement` class is introduced. For a reified statement, an instance of `rdf:Statement` is created. The properties `rdf:subject`, `rdf:predicate` and `rdf:object` define the subject, predicate or object of the statement respectively.

RDF provides different containers and collections. A container defines an *infinite* group of things whilst a collection defines a *finite* group of things. That means that collections are closed [23]. The members of a collection are complete, no other members exist. A collection in RDF is represented by the `rdf:List` class. A `rdf:List` is constructed using the properties `rdf:first` and `rdf:rest`, and the resource `rdf:nil`. `rdf:first` points to the first element in the list, `rdf:rest` to another list containing the rest of the list. Figure 2.3 shows an example graph that represents the sentence "Sebastian has written the papers A, B and C." Each blank node forming this list is implicitly of type `rdf:List` [23], so it is not necessary to state that explicitly in a statement. The last list item's `rdf:rest` property points to `rdf:nil`, a predefined resource representing an empty `rdf:List`.

In addition to the `rdf:List` class, RDF defines the following three types of containers:

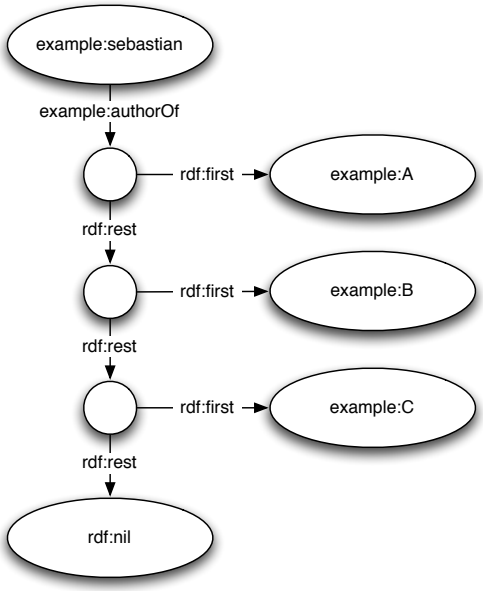


Figure 2.3: Example RDF graph representing a list collection.

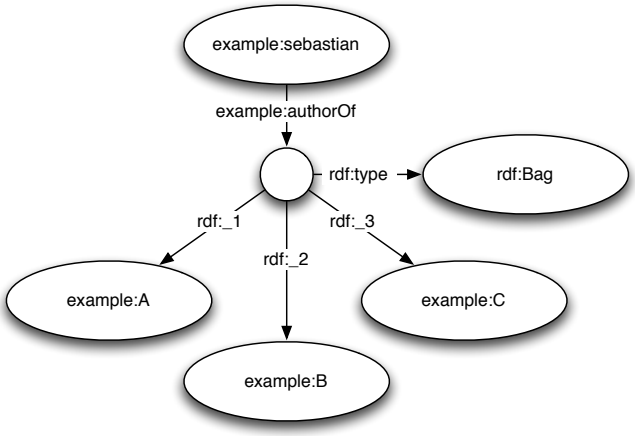


Figure 2.4: Example RDF graph representing a bag container.

- `rdf:Bag`
- `rdf:Seq`
- `rdf:Alt`

A *bag* is an unordered group of resources or literals with potentially duplicate members. A *sequence* is a group of resources or literals where the order is significant, possibly containing duplicate members. An *alternative* represents a group of resources / literals that are alternatives, i.e. an application can use an arbitrary member of this group. For example, it can describe a list of alternative internet sites where a file can be downloaded. All these containers are described by one resource (or blank node) representing this list. This resource has one of the three types. The items of the list are linked with container membership properties. They are named `rdf:_n`, where `n` is an integer greater than zero without leading zeros, e.g. `rdf:_1`, `rdf:_2`, etc. A collection representation of the list shown in figure 2.3 is depicted in figure 2.4.

RDF also defines a class of special literals, `rdf:XMLLiteral`, which is a literal that contains XML markup.

Please refer to the W3C RDF primer [23] for more details about the RDF vocabulary.

Serialisation

To make a RDF model machine-readable, a standard syntax for RDF models is needed. In fact, there are many different ways for the serialisation of RDF models like RDF/XML, which serialises RDF in XML. Other syntaxes like N-Triples or Notation 3 (N3) have been developed for either better readability for humans (N3) or for effective parsing (N-Triples). A sample RDF/XML serialisation looks like this:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/1.0/">
  <rdf:Description rdf:about="http://eric.com/foaf.rdf">
    <foaf:knows>
      <rdf:Description
            rdf:about="http://steve.com/foaf.rdf"/>
    </foaf:knows>
  </rdf:Description>
</rdf:RDF>
```

An N3 serialisation of the same triple looks as follows:

```
@prefix foaf: <http://xmlns.com/foaf/1.0/> .

<http://eric.com/foaf.rdf> foaf:knows
     <http://steve.comfoaf.rdf/> .
```

N-Triples is very similar to N3, they both serialise one triple per line. N3 additionally allows the abbreviation of namespace prefixes and the omission

of the subject if some triples have the same subject and are serialised in neighboured rows in the file. See [5] for more details about N3. Turtle [4] is another RDF syntax, it is a subset of N3.

2.1.3 RDF Schema

RDF schema (RDFS) is RDF's vocabulary description language. It provides a type system for RDF, similar to type systems in object-oriented programming languages. Resources can grouped together using classes. Members of a class are called *instances*. The `rdf:type` property assigns a class to a resource. The group of RDF schema classes is itself a class called `rdfs:Class` [7]. The *class extension* of a class is the set of instances of this class. The `rdfs:subClassOf` property defines a subclass relation. RDF schema defines the following classes:

rdfs:Resource is the class of all things described by RDF.

rdfs:Class is the class of resources that are RDF classes.

rdfs:Literal is the class of literal values. Literals may be plain or typed. A typed literal is an instance of a datatype class. `rdfs:Literal` is an instance and a subclass of `rdfs:Class`.

rdfs:Datatype is the class of datatypes. `rdfs:Datatype` is an instance and subclass of `rdfs:Class`. Each instance of `rdfs:Datatype` is a subclass of `rdfs:Literal`.

rdf:Property is the class of RDF properties.

RDF schema provides different properties to define hierarchies of classes and properties, to define domain and range of properties and to add labels and descriptions to resources. The following properties are defined by RDF schema:

rdf:type is used to state that a resource is an instance of a class.

rdfs:subClassOf is a transitive property that defines subclass relations. If class A' is a subclass of class A, instances of A' are also instances of A.

rdfs:subPropertyOf is a transitive property that defines subproperty relations. If P' is a subproperty of P, all resources related by P are also related by P'.

rdfs:domain is used to state that a resource that has a given property is an instance of one or more classes.

rdfs:range is used to state that the values of a property are instances of one or more classes.

rdfs:label is used to provide a human-readable representation of a resource.

rdfs:comment is used to provide a human-readable description of a resource.

In contrast to type systems in object-oriented programming languages, RDF schema does not define valid attributes for instances of classes but it defines class instances for attributes. A complete specification of the RDF schema vocabulary can found in [7].

2.2 Semantic Wikis

Semantic wikis combine the easiness and versatility of wikis with the idea of an Semantic Web. They allow collaborative creation of semantically annotated content without any restrictions on the domain. In most implementations, wiki text and semantic annotations can be mixed arbitrarily. There are many different implementations which vary in their use case and their maturity. Well-known semantic wiki implementations are Semantic Mediawiki[3] [38], a semantic extension of Wikipedia's wiki engine Mediawiki, SemperWiki[4] [27, 30], a semantic personal wiki and IkeWiki[5] [34]. The advantage of total freedom on the described domain makes it harder to support the user creating annotations. All mentioned implementations either do not provide annotation support or they only list all known predicates.

2.3 ActiveRDF

Due to the decentralised infrastructure of the Semantic Web, the knowledge is distributed over multiple data sources. Most current RDF application programming interfaces (API) provide a triple-centric access to different data sources. Data sources in the Semantic Web can be heterogeneous: they can be relational databases, HTML pages with embedded RDF, SPARQL endpoints or plain RDF files. Most APIs interact with these data sources using RDF-specific classes, like e.g. *Triple* and *Graph*. This triple-centric access might confuse users that are not familiar with RDF and prevents them from using RDF for storing and exposing data.

ActiveRDF[6] [29] is a RDF library for the Ruby programming language. It provides a more intuitive, object-oriented access to RDF data which hides the complexity of RDF. The programmer using ActiveRDF does not have to handle different RDF serialisations, graphs and triples. Instead, he can use a domain specific language (DSL) to programmatically access RDF data.

[3] `http://ontoworld.org/wiki/Semantic_MediaWiki`
[4] `http://www.semperwiki.org`
[5] `http://ikewiki.salzburgresearch.at/`
[6] `http://www.activerdf.org/`

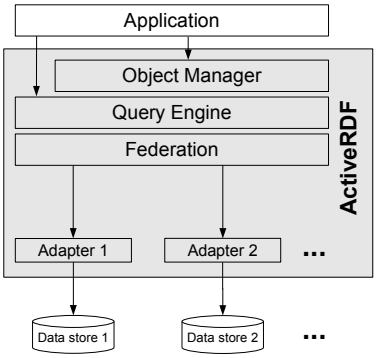


Figure 2.5: Architecture of ActiveRDF

ActiveRDF can compared to object-relational mappers: they provide object-oriented access to a relational database. Unlike relational databases, Semantic Web data has no fixed structure, it is dynamic. Thus, a static mapping as used in object-relational mappers is not completely sufficient. A dynamically typed programming language like Ruby or Python is more suited to this dynamic structure than statically typed programming languages like Java, C or C#. In this section, we first present the architecture of ActiveRDF. Then we explain its different components. Finally, examples for different use cases of ActiveRDF are given.

2.3.1 Architecture

The general principle of ActiveRDF is the representation of a RDF resource through transparent proxy object. Read and write methods on the proxy object are directly forwarded to the RDF model. The proxy object does not store any state. Figure 2.5 shows an overview of the architecture of ActiveRDF. ActiveRDF consists of four components which abstract from RDF to objects gradually.

Object manager

The object manager provides a *virtual* API for the programmer. It is called *virtual* because the classes and methods are not generated. Unhandled method calls are caught by using meta-programming and then translated into RDF queries. The object manager maps objects to RDF resources, classes to RDFS classes (if a RDF schema exists) and methods to data manipulation queries. For example, when the application calls a *find* method, the object manager translates this operation into a query on the data source. When a resource belongs to multiple classes and two or more classes define the same method, a resolution strategy must be used to choose a method. Various possibilities exist to handle this multiple inheritence: The first

method found can be used, an error can be raised or the user can indicate a preference for certain methods.

Query engine

The object manager uses to the query engine to construct queries from method calls. The query engine provides an query API that is independent of a specific data source and query language. The query engine returns an object representing a query which is translated to a data source specific query by the appropriate adapter.

Federation Manager

The federation manager handles the integration of multiple data sources. It distributes queries to some or all these sources and collects the query results. In the current implementation, queries are distributed to all sources subsequently.

Adapters

Adapters are the data source specific component of ActiveRDF. They translate the query objects to database compliant queries and execute them. Each adapter must implement only a few methods which simplifies writing an adapter.

2.3.2 Usage

We now show a few examples to demonstrate ActiveRDF's features. Please refer to the wiki[7] of the ActiveRDF project for a detailed documentation of its usage.

Connecting to a data source

To support multiple data sources, ActiveRDF maintains a connection pool to these sources. The developer can add and remove data sources to and from the pool. In the following example, a connection to a SPARQL endpoint is established.

```
ConnectionPool.add(:type    => :sparql,
                   :results=> :sparql_xml,
                   :url     => 'http://browserdf.org:81')
```

In addition to the type of the connection, the URL to the data source must be given. Optionally, we can specify the desired result format. Most parameters are adapter dependent, e.g. an URL parameter for an in-memory triple store does not make sense. In the next example, we connect to a RDFlite

[7] http://wiki.activerdf.org

data store. RDFlite is an embedded triple store based on SQLite[8]. It provides both read and write operations, and, with an additional Ruby library, fulltext search. Now we connect to a RDFlite data store which is stored in a file.

```
ConnectionPool.add(:type     => :rdflite,
                   :location=> 'foaf.dat')
```

For a full reference over the existing adapters and their parameters, please refer to the ActiveRDF wiki.

Data manipulation

Now that the connection to one or more data sources has been established, we can access and manipulate the data. Assume that the RDFlite data store we are connected to contains FOAF data, including the FOAF schema. Now we want to add a person to this store. First, we advise ActiveRDF to register the FOAF Namespace and to generate the classes for the FOAF schema.

```
Namespace.register(:foaf, 'http://xmlns.com/foaf/0.1/')
ObjectManager.construct_classes
```

Now we can create a new Person, add some properties to it and save the changes.

```
sebastian = FOAF::Person.new('http://www.deri.org/gerke')
sebastian.firstName = 'Sebastian'
sebastian.surname = 'Gerke'
sebastian.mbox = 'sebastian.gerke@deri.org'
sebastian.save
```

Dynamic Finders

When we want to lookup a person's mail address, we use ActiveRDF's `find` methods. We can either find all resources of a certain type or we can find resources that match an attribute that we are looking for. The following listing shows how to find all persons in the data store.

```
all = FOAF::Person.find_all
all.each do |person|
  print person.surname
end
```

If we are only looking for persons with e.g. e certain surname, we use the `find_by` methods that ActiveRDF's virtual API provides.

```
sebastian = FOAF::Person.find_by_surname('Gerke')
print sebastian.mbox
```

Now we retrieved the mail address of Sebastian.

[8] `http://www.sqlite.org`

Query engine

For some purposes, using the virtual API may not be sufficient or inefficient. For advanced queries, ActiveRDF exposes its internal query engine. A simple query for all subjects in the triple store looks as follows.

```
query = Query.new
resources = query.select(:s).where(:s,:p, :o).execute
```

This query finds all subjects, ignoring dupllicates. A distinct list of all subjects is obtained by using the `distinct` method instead of the `select` method.

```
query = Query.new
resources = query.distinct(:s).where(:s, :p, :o).execute
```

To obtain the number of resources that match a certain criteria, the `count` method is used. The following query finds the number of persons whose surname is "Gerke". Multiple criteria can be used in a query by using the `where` method repeatedly.

```
query = Query.new
query.count.distinct(:s)
      .where(:s,
             Namespace.lookup(:rdf, :type),
             Namespace.lookup(:foaf, :person)
      .where(:s,
             Namespace.lookup(:foaf, :surname),
             'Gerke')
```

Chapter 3

Personalised Wiki Syntax

Currently, many different wiki systems exist with often many different wiki syntaxes, making it difficult for users of different wikis to use the right wiki markup elements. An experienced user of one wiki would often make mistakes when editing a different wiki system. These mistakes are time-consuming: users need to manually replace wrong syntax elements (since browsers do not support search and replace in text fields) or copy the text into a text editor, edit it, and copy it back into the wiki.

For users of different wikis, it would be comfortable if each user can choose his own favourite wiki syntax. If every wiki would support such a personalisation of wiki syntax, a user can create a his own preferred syntax description and use this syntax for all wikis he uses. Such a personalisable wiki syntax is described in this chapter.

Several challenges need to be overcome to enable a personalisable wiki syntax. First, we need to store the wiki content in a syntax independent manner. Simply storing the markup of the wiki content, like `*`, `==`, as used in most current wikis, is not feasible since the markup would have different meaning for different users. A wiki data model as an intermediate representation of content is needed. Therefore the wiki markup has to be parsed into this wiki data model. This reveals the second challenge: An fault-tolerant parser that parses all recognised wiki syntax elements and treating everything that is not recognised as plain text. This challenge occurs only if a "real" parser is used. The search-and-replace parsing used in most wiki engines does not have this problem because it only replaces recognised elements and leaves the rest unchanged. Thirdly, the parser must be able to change the tokens at runtime. This implies that the parser either must not be generated by grammar. It must be completely dynamically configurable.

The structure of the chapter is as follows: First, a syntax independent data model for wiki content is presented in this chapter. Then the architecture of the wiki syntax engine (for rendering and parsing) is described.

Finally, a description and discussion of the wiki syntax parser follows.

3.1 Semantic Wiki Data Model

To allow a syntax-independent storage of the wiki content, we need a data model for this content. Most current wikis store the markup directly and render the content to HTML at access time by replacing the wiki markup with HTML tags by means of regular expressions. This approach is not feasible for a wiki that allows a customisable syntax. To allow such a customisable syntax, an intermediate data model is necessary. There are several existing approaches for a wiki data model, namely flexbisonparser[1], an extension to the Mediawiki engine, wiki interchange format (WIF)[2], WikiWyg[3], WikiModel[4], InterWikiMarkupLanguage (IWML)[5] and IkeWiki[6]. Since none of these models have yet emerged as an accepted standard and since for our prototype a minimalistic model would be sufficient, we designed a minimal Semantic Wiki model, shown in Figure 3.1.

The wiki data model is a page centric data model, additional data and features such as user and rights management has not been included in the data model. The content of the wiki is represented by the `Element` class. This class is the superclass of all elements that appear in a wiki page. Some of these elements can be containers for other elements, they are described by the `Container` class. Concrete subclasses of this class are classes that represent a certain text style, namely *italic*, *bold*, *teletype*, *underline* and *strikethrough*. These containers also store their content in an ordered list. Elements that are not containers cannot embed arbitrary syntax elements. These elements are *links*, *images*, *headers*, *text*, *end of line*, *begin of line* and semantic *annotations*. A *list* is a more complex syntax element, it consists of an ordered list of *list items* which can contain arbitrary elements. There are two types of lists: ordered lists and unordered lists. Here is a detailed description of all container syntax elements, which are used for text formatting:

Italic Prints the content in *italic* type.

Bold Prints the content in **bold** type.

Underline Prints the content underlined.

Teletype Prints the content in a `typewriter` font.

[1] `http://svn.wikimedia.org/svnroot/mediawiki/trunk/flexbisonparse/`
[2] `http://ontoware.org/projects/wif/`
[3] `http://www.wikiwyg.net/`
[4] `http://wikimodel.sourceforge.net/`
[5] `http://www.altheim.com/specs/iwml/`
[6] `http://ikewiki.salzburgresearch.at/`

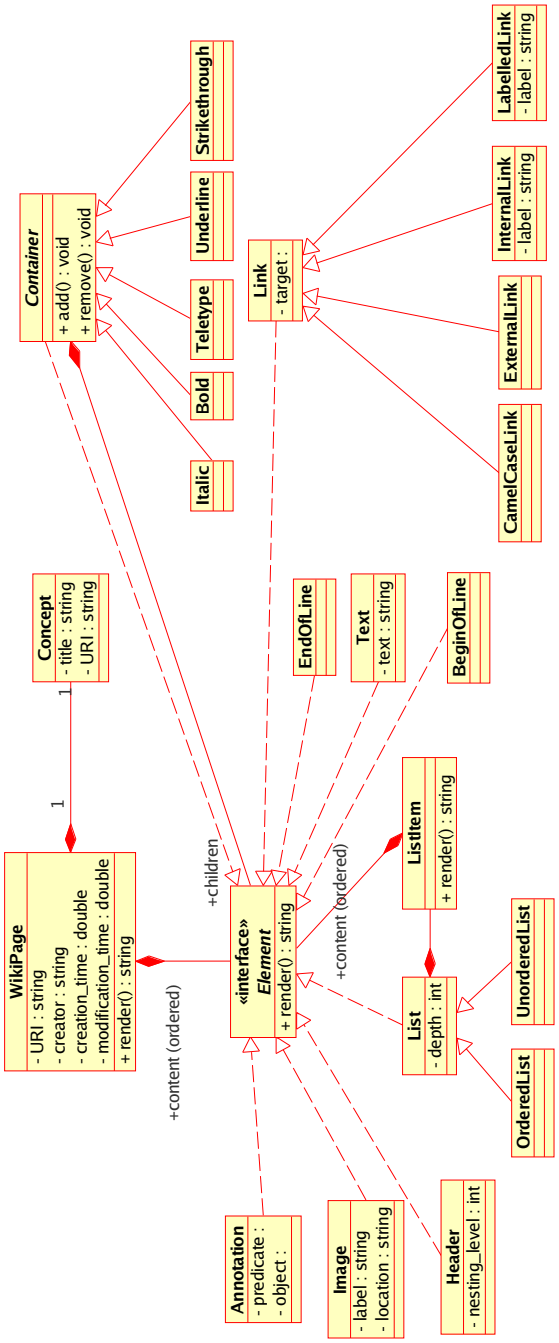


Figure 3.1: Wiki Model

Strikethrough ~~Strikes through~~ the content.

And here a description of basic elements that are not containers:

Link A hyperlink to another page. Targets can be external or internal, and they have an optional label.

Header A header of a section. A nesting level is stored with the header to assign a correct formatting to the header.

Image An image in a wiki page. It contains the URL where the image can be found. For simplicity we do not specify more advanced image properties like size, alternative text.

Annotations Allows to semantically annotate the concept of the current wiki page. The predicate and object can be assigned by the user, the subject is automatically assigned by the system to the concept of the current page.

Text Plain text.

End Of line Special element for the end of a line.

Begin of line Special element for the begin of a line.

List A special case is a list, it contains list items, which are containers for arbitrary content. A list can either be ordered:

1. Red
2. Green
3. Blue

or unordered:

- Red
- Green
- Blue

One major design decision is the separation of a wiki page from the concept that is described by this wiki page. This allows statements about both a wiki page and the concept described by the wiki page without confusing these two. Imagine a Wikipedia page about the book "Das Parfüm". In a wiki that does not differ between the wiki page and the concept it describes, a statement like `dc:creator 'Patrick Süsskind'` can be interpreted in two ways: Either Patrick Suesskind is the author of the book, or he is the author of the Wikipedia article about that book. A separation between the wiki page and the concept described by a wiki page avoids this ambiguity. So each `WikiPage` has one `Concept` that it describes. Additionally, some intrinsic metadata like creation time, author, modification time is maintained for a wiki page.

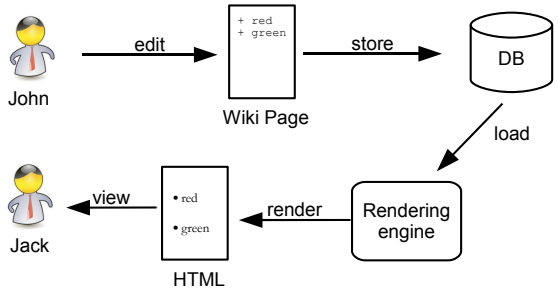


Figure 3.2: Edit / View process of a standard wiki engine (without syntax customisation)

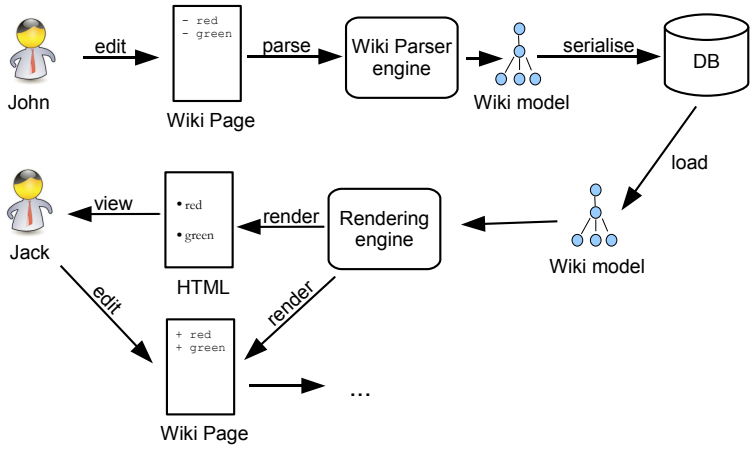


Figure 3.3: Edit / View process of a wiki engine with syntax customisation

3.2 Wiki Engine Architecture

The architecture of the wiki rendering and parsing engine differs from those from traditional wikis as it uses an intermediate data model for the page content instead of storing the wiki markup directly. This data model can be either serialised into XML or JSON or in any other serialisation format. Figure 3.2 illustrates the edit and view process of a conventional wiki. It shows a typical wiki with one user editing a wiki page and another user viewing this wiki page. John creates a wiki page by adding a list with two elements. This page is directly (without any processing step) stored into the database when John submits the edited page. When Jack wants to see that page, it is loaded from the database and then rendered into HTML by regular expressions search and replace.

Figure 3.3 shows an overview of a similar process for our wiki with customisable syntax. Again, John creates a wiki page containing a list with two items. Then the wiki markup text is parsed into the wiki data model by the

wiki parsing engine. Subsequently, this intermediate model is serialised into XML or any other serialisation (e.g. any RDF serialisation format, JSON). Of course, for a Semantic Web application, a RDF serialisation seems to be the best option. The problem is that RDF is not efficient for storing long ordered lists which can make such an implementation not very well-performing for a highly used wiki. Now that the content of the recently added page is stored, it is available for viewing and editing for other users. If Jack wants to view and edit this page, the data model for that page is loaded from the database. The rendering engine transforms this model into either HTML for viewing or into Jack's favourite wiki syntax for editing.

3.3 Parsing Wiki syntax

As explained in the previous section, we need a "true" parser to transform the wiki markup into a wiki data model.

Firstly, we explain the language theoretic aspects of wiki syntax. We explain we existing Ruby parser generators are not sufficient for parsing wiki syntax. We evaluated some Ruby parser generators if they support unbound lookahead. All evaluated Ruby parser generators, namely Racc[7], a LALR(1) parser generator, and Coco/R(uby)[8], a LL(1) parser generator, do not support unbound lookahead. Thus a new lexer and parser had to be created. Primary design goals of this parser are:

- Support for most common wiki syntax constructs
- Customisation of tokens
- Parse as much as possible (error tolerant parsing)

Especially the second and third aspect made the utilisation of parser generator more difficult: Most parser generator need a grammar file containing the tokens to parse. They do not allow to specify the token after creating a parser. Modifying a generated parser could be an option, but this approach is not very clean and the fact that we need an unlimited lookahead for lexical analysis makes the usage of existing lexer / parser generators impossible. Thus we decided to handwrite a lexer and a parser to parse wiki syntax with with customisation support.

3.3.1 Wiki Syntax

Wiki Syntax in general is not very complex. There are a few basic types of syntax constructs and most wiki markup elements use one of these basic types with different tokens. These basic syntax constructs are:

[7] http://i.loveruby.net/en/projects/racc/
[8] http://www.zenspider.com/ZSS/Products/CocoR/

- Text delimited by one separator for both start and end of an element. Example: `__underline__`. This construct is mostly used for text formatting.
- Text delimited by two different separators for start and end of an element. Example: `[Link]`. This construct is often used for hyperlinks.
- Text in one line that starts with a certain token. Example:
  `* list item`
  This construct is mainly used for lists and headers.

These are the basic concepts that are used for wiki syntaxes. Further specialisation of these concepts such as labelled links (`[WikiPage - a good page]`) or macros can be modelled by firstly using the predefined concepts and then parse the content of the link by regular expressions or with simple string manipulation methods.

Error tolerant parsing of wiki content means that all text that is entered into the wiki has to be parsed, even when it does not contain any correct wiki markup or all opened tags are not closed. Parts of the input string that is not recognised as wiki markup keywords are then parsed as text. This requirement makes parsing wiki markup so difficult: it is not possible to determine the meaning of a token (if it is a token or plain text) without looking at the rest of the input character stream. A similar problem is described in detail in [3]. The following small example demonstrates the problem: Imagine a wiki where text between '`[`' and '`]`' is parsed as a hyperlink. Now the input string "`This is [ no link`" is parsed character by character. When reaching the character '[', we can not determine if it is the opening token of a hyperlink or plain text without any knowledge about the rest of the input string.

There are numbers of different approaches for dealing with context-dependent tokens [3]. The authors describe in this article an approach for parsing with context-dependent tokens. The basic idea is to store all possible tokens for a given lexeme during the lexical analysis and later (at parsing time) decide which token to choose. That means that the lexical analysis produces a sequence of token sets instead of a sequence of tokens.

We develop a context-dependent parsing technique tailored for parsing wiki markup. The presented algorithm is similar to [3]. The main difference is that in the present case, we do not need to store multiple possible tokens for each lexeme because each lexeme in a wiki can have exactly two tokens: A special wiki token or a text token. Because all lexemes can also produce a text token, a separate storage of this knowledge is not necessary. We only store a token sequence containing special wiki tokens. Whenever we encounter that a special wiki token is not valid, we transform it into a text token.

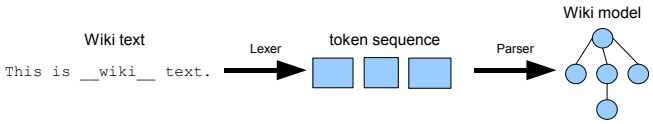
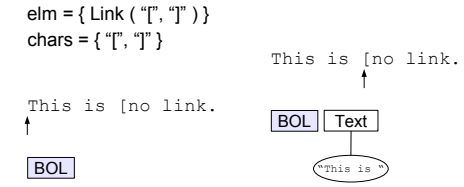
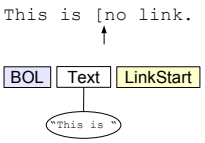


Figure 3.4: Overview of the parsing process. Firstly, the so-called lexer transforms a character sequence into a shorter token sequence. Then the parser transforms this token sequence into a parse tree.

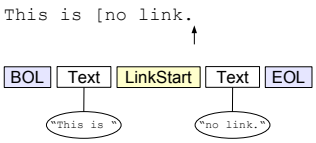


(a) Sets *chars* and *elm* with sample input string before begin of lexing process.

(b) Step 1



(c) Step 2



(d) Step 3

Figure 3.5: Sample lexer run with input string and generated token sequence. Arrow marks the current lexer position.

3.3.2 Parsing Process

An important feature of wiki markup is that all characters that are not recognised as keywords are treated as text. Furthermore, it is more unlikely that opened syntax elements (brackets) are not closed than they are closed. These preconditions can help to build an efficient parser. The basic idea is to build a standard lexer and parser with a small extension: If, at the end of the input string, a opened syntax element is not closed, a small (constant time) tree operation on the parse tree is used to convert the wrongly parsed wiki element into a plain text element. This tree conversion is basically a simple node deletion followed by a possible merge of neighboured nodes if one or two neighbours are also text nodes. Overlapping syntax elements are not supported, the first syntax element is parsed correctly but the second one is parsed as plain text.

See figure 3.5 and 3.6 for an example lexer and parser run for the input string *"This is [no link"*. Before lexing, some initialisation has to be done: First, the syntax has to be configured, i.e. the tokens for the syntax elements

are passed to the lexer along with their meaning. In the example, a *Link* is the only wiki syntax element used. A link is delimited by `[` at the begin and by a `]` at the end. The definitions for all syntax elements are stored in the set *elm*. The lexer then checks the tokens for all single characters that are used in all tokens. These characters are stored in the set of special characters *chars*. Now the lexing process begins. First, a begin-of-line token is written to the output token stream. Then the first character '`T`' of the input string is read. As '`T`' it is not in the set of special characters, the input string is read until a special character or a line break is found. The whole substring '`This is `' is stored in a new text token and written to the output stream. The next character '`T`' is in the set of special characters. Though a token for the start of a link is written to the output stream. Then, the rest of the text is written as one text token to the output stream. As the end of the input string is reached, an end-of-line token is finally written to the output stream.

Now this token stream can be parsed, i.e. transformed into a parse tree. For the first two tokens, the begin-of-line token and the first text token, corresponding nodes are added to the tree (See figure 3.6a). The next token is the link start token. Now a link node is added to the parse tree and this node becomes the current node. A pointer to the link node is pushed on top of a stack that holds all opened syntax elements. Then a node for the start delimiter `[` is added as a child of the link node (Figure 3.6b). The next text token and the end-of-line token are also added as children of the link node (Figure 3.6c). Now that the end of the input token stream is reached and no closing token for the link is found, the link node has to be removed from the parse tree. Therefore the link node itself is removed from the parse tree and all its children are added as children of the parent node of the removed link node (Figure 3.6d). Finally, neighboured text nodes are merged into one text node (figure 3.6e). Now the parsing process is finished and the wiki markup text has be transformed into a wiki model. In the next two sections, the implementation of the lexer and the parser are explained more detailed.

3.3.3 Lexical Analysis

Lexical analysis is the processing of a input sequence of characters into a sequence of logical units, so-called tokens. This token sequence is then usually forwarded as input to a parser. Usually, the lexical analyser is generated by a lexer generator. In the presented system, the lexer is handwritten due to the need for a configurable syntax. As explained in section 3.3.1, the meaning of a token cannot be determined without an arbitrary lookahead. All tokens have either a special meaning (like start token for a header or hyperlink), or they are plain text. To overcome this problem, each possible token is treated as a special token by the lexical analyser. Only the parser decides if it really is a special token, otherwise it is converted to a text to-

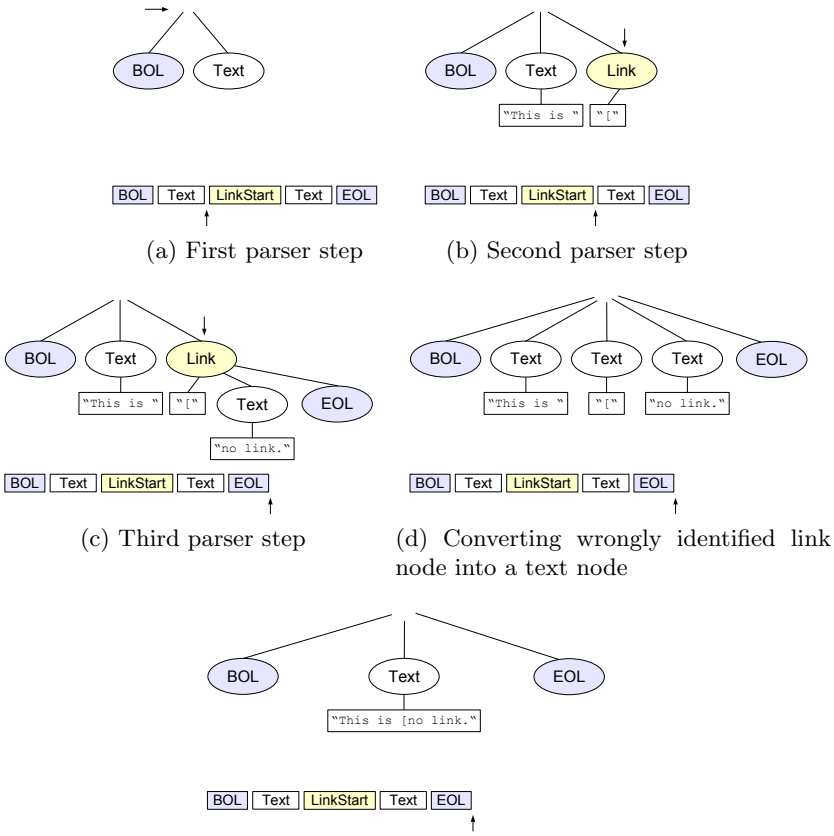


Figure 3.6: Sample parser run with generated syntax tree and input token stream (bottom). The arrow at the syntax tree marks the current node, the arrow at the token stream marks the current position in the token stream.

Listing 3.1: Lexer class

```
class Lexer
  $TEXT = :Text
  $BOL  = :BOL
  $EOL  = :EOL

  def initialize()
    @symbol_table = Hash.new
    @symbol_characters = Set.new
  end
end
```

Listing 3.2: Token class

```
class Token
  attr_accessor :type, :lexeme
end
```

ken. Now a detailed description of the lexical analysis algorithm follows. The code examples are written in a ruby-like pseudo-code.

The lexical analyser is implemented in the `Lexer` class (see listing 3.1). First, the lexer class is initialised: Three standard token types that are independent of the syntax are defined: The *text* (`Text`) token type, the *begin of line* (`BOL`) token type and the *end of line* (`EOL`) token type. A symbol table that maps lexemes to token types and a set of symbol characters are initialised.

A token consists of a type and a lexeme (see listing 3.2). To add a new user-defined token to the lexer, the `add_node` method that allows to define the lexeme and the type of this token is defined in the Lexer class (see listing 3.3). For each added token, an entry in the symbol table that maps from the lexeme to the token name is created. Each lexeme's character is added to the set of symbol characters. The method `lexical_analysis(input)` starts the lexical analysis of the given input string. First, a lexical analysis is performed, which returns a token sequence. As shown later in this section, the token sequence can contain multiple text tokens in a row. In

Listing 3.3: Method for adding new tokens to the lexer

```
class Lexer
  def add_token(lexeme, name)
    symbol_table[lexeme] = name
    for char in lexeme do
      symbol_characters.add(char)
    end
  end
end
```

Listing 3.4: Wrapper method that performs the lexical analysis and compresses the token stream

```
def lexical_analysis(input)
  token_sequence = compress(lex(input))
  return token_sequence
end
```

Listing 3.5: Lexer method

```
1  def lex(input)
     token_sequence = []
     token_sequence << Token.new($BOL, '')

     i = 0
6    while i < input.size
       current = input[i].chr
       if current == "\n":
         token_sequence << Token.new($EOL, "\n")
         token_sequence << Token.new($BOL, "");
11     elsif symbol_characters.include?(current)
         token_cand = get_keyword_string(input[i, input.size])
         token = find_token(token_cand)
         i += token.lexeme.size - 1
         token_sequence.add(token)
16     else
         text = get_text_token(input[i, input.size])
         i += text.lexeme.size - 1
         token_sequence.add(text)
       end
21     i += 1
     end
     token_sequence.add(Token.new($EOL, "\n"))
     return token_sequence
   end
```

the `compress` method, these consecutive text tokens are merged into one text token where the lexemes of the original tokens are concatenated. This merge is not necessary but it keeps the wiki data model more consistent to the wiki markup. The lexical analysis itself is done in the `lex` method (listing 3.5). First, the output token sequence is initialised with a begin of line token (line 3). Then, each character from the input string is read in a loop (line 6 - 7). If the current character is a linebreak, an end of line token and a begin of line token is added to the token sequence (lines 8 - 10).

If the current character is in the set of symbol characters, it is possibly the start of a keyword (line 11). In this case, the lexer tries to find a keyword (line 12). The method `get_keyword_string` searches the longest substring starting from the current character that contains only special characters. This token candidate is then checked if it is token (line 13). In this case, a new token instance of the according type is created, otherwise a text token

is created. If a text token is created, it is possible to have multiple text tokens in a row, which makes the `compress` method mentioned earlier recommendable. Then the new token is added to the token sequence (line 15).

If the current character is neither a newline character nor a special character, it is treated as text. Therefore all characters that are neither newline characters or special characters are read from this position and a new text token is added to the token sequence (lines 17 -19).

Now that the input character sequence is transformed into a token sequence, the parser can transform it into a parse tree, as explained in the following section.

3.3.4 Parser

To obtain a syntax tree that represents the intended structure of the wiki markup, the token sequence has to be transformed from the token sequence into the desired syntax tree. In the present case, the syntax tree is represented by the wiki data model. As explained in section 3.3.1, the system supports three major syntax concepts. These concepts are:

- Syntax elements with same token for start and end. They are called *symmetric syntax elements*.
- Syntax elements with different tokens for start and end. They are called *asymmetric syntax elements*.
- Syntax elements that span a whole line and start with a certain token. They are called *line syntax elements*.

The parser then can be configured to parse an arbitrary amount of these syntax elements. Each of these basic syntax elements is defined by its name and lexemes, one lexeme for symmetric and line syntax elements, two lexemes for asymmetric syntax elements. In the initialisation phase, the parse tree with a root node is created and the current node pointer is set to the root node. Furthermore an empty stack for opened syntax elements is instantiated and two boolean flags, the begin-of-line (`bol`) and the special-line flag, are set to false. The special-line flag is set if the current line is in a line syntax element. The begin-of-line flag is set if the previous token was a begin-of-line token. The code in listing 3.6 shows the complete parsing algorithm.

The parsing process starts in line 5. Each token is read subsequently. If the current token is a begin-of-line token, the begin-of-line flag is set (line 7) and the next token is read. Otherwise, if the begin-of-line flag is set, a begin-of-line node is added to the parse tree as a child of the current node (line 9). If the current token is a text token, a new text node is added to the current node (line 12 - 13). If the current token is an end-of-line token and the special-line flag is set, the corresponding opening token is searched

Listing 3.6: Main parser method

```
def parse_token_stream (token_stream)
  bol = false
  special_line = false
  for token in token_stream
    if (token.type == $BOL)
      bol = true
    else
      if bol
        add_text_node(BOLNode.new())
      end
      if token.type == $TEXT
        text_node = TextNode.new(token.lexeme)
        add_text_node(text_node)
      elsif token.type == $EOL
        if special_line
          find_line_opening_token_and_cleanup()
          special_line = false
        else
          add_text_node(EOLNode.new())
        end
      else
        element = get_syntax_element_for_token(token)
        if element.instance_of?(SymmetricSyntaxElement)
          position = search_syntax_element_in_stack(element)
          if position == -1
            add_node(Node.new(element))
          else
            find_opening_token_and_cleanup(element)
          end
        elsif element.instance_of?(LineSyntaxElement)
          if bol
            special_line = true
            add_node(Node.new(element))
          else
            add_text_node(TextNode.new(element.token))
          end
        else
          if is_start_token(token)
            add_node(Node.new(element))
          else
            position=sarch_syntax_element_in_stack(element)
            if position == -1
              add_text_node(TextNode.new(element.end_token))
            else
              find_opening_token_and_cleanup(element)
            end
          end
        end
      end
      bol = false
    end
  end
  cleanup()
  return @root
end
```

Listing 3.7: Method for adding a node into the syntax tree

```
def add_node(node)
  node.parent = @current_node
  @current_node.children.add(node)
  @stack.push(node)
  @current_node = node
end
```

Listing 3.8: Method that searches the stack for a specified syntax element

```
def find_opening_token_and_cleanup(syntax_element)
  while !@stack.empty?
    node = @stack.pop()
    if node.syntax_element == syntax_element
      @current_node = node.parent
      return
    else
      convert_to_text_node(node)
    end
  end
  if @stack.empty? &&
     syntax_element.instance_of?(AsymmetricSyntaxElement)
    text_node = TextNode.new(syntax_element.end_token)
    text_node.parent = @current_node
    @current_node.children.add(text_node)
  end
end
```

on the stack and all elements above the searched token in the stack are converted to text nodes. That means that all opened syntax elements that are not closed in a special line are treated as text (line 16). The special-line flag is unset (line 17). If the special-line flag is not set, an end-of-line node is added to the parse tree (as a child of the current node) (line 19).

If the current token is a opening or closing token of a syntax element (line 21), it is checked if it is a token for a symmetric syntax element (line 23). In that case, the stack is searched for a node of this syntax element. If no node is found (line 25), a new node is added to the syntax tree, even if it is not sure that an closing token will be found later. This is done by the `add_node` method shown in listing 3.7. To add a new node to the syntax tree, the new node's parent is set to the current node, it is added to the children of the current node and it is pushed onto the stack. Finally, the new node becomes the current node.

If an opening node is found (line 28), `find_opening_token_and_cleanup` is called (see listing 3.8. This method searches the stack from top for the specified syntax element and converts each node that is over the searched node to a text node. If the node is found, its parent becomes the current node. in the case that the stack is empty and the syntax element is an asymmetric syntax element, i.e. a corresponding opening node could not be

Listing 3.9: Cleanup method

```
def cleanup()
  while !@stack.empty?
    n = @stack.pop()
    convert_to_text_node(n)
  end
end
```

found, the closing token is converted to a text node and this text node is added to the parse tree.

If a token of a line syntax element is found (line 30), the begin-of-line flag is checked. If it is set, the special-line flag is set and a new node is added to the parse tree by calling the `add_node` method described earlier in this section. If the begin-of-line flag is not set, a text node containing this token is added to the syntax tree (line 35).

If the token is the start token of an asymmetric syntax element, a new node of that syntax element is added to the syntax tree (line 39). If it is the end token, the stack is searched for the corresponding syntax element and if none is found, a text node containing the closing token is added to the parse tree. If the corresponding node is found, all nodes above this node in the stack are converted to text nodes (line 45). If the current token is no begin-of-line token, the begin-of-line flag is unset. Then the `cleanup` method is called (listing 3.9). It converts all nodes on the stack to text nodes. Finally, the complete syntax tree is returned.

This section presented algorithms for parsing arbitrary Wiki syntax with customisable syntax elements and tokens. Existing parser generators could not be used for two reasons: First, parser generators are static, they generate the parsers, thus some kind of reflection would be necessary to allow customisable tokens. Secondly, existing parser generators (in Ruby) do not allow arbitrary lookahead, which is necessary to determine if a token has to be parsed as a token for wiki syntax or if it has to be parsed as text. We therefore manually implemented a personalisable parser for wiki markup. The lexical analyser of this parser treats each lexeme as a potential special wiki token. The parser then decides if this token is correct, otherwise each incorrect token is converted to a text token.

The complete parser's source code can be browsed online on the wiki project site[9] or downloaded[10].

[9] `http://code.launchpad.net/~sebastian-gerke/swee/semperwiki`
[10] `http://m3pe.org/~sebastian/wiki_model.zip`

Chapter 4

Reusing Desktop Data

Much data, especially for personal information management, is already available on the desktop. If a user wants to refer to this data, he has to re-enter all the metadata. Therefore, an import function for common desktop data like e-mails, instant messaging data, text documents, bibtex files, calendar data and address book data would be a benefit. Users could drag and drop data from the desktop and from other applications into their semantic authoring application and all file metadata can be either stored in the application or linked so that a change in that data would also be visible for the user in his application.

Imagine we want to add a comment about a paper: we might want to annotate that it belongs to a certain project, that response is due on a certain date or that we like it. To refer to this paper, we currently have to describe and annotate it with metadata like title, author, creation date: metadata that already exists in the document itself.

In this chapter, we propose an architecture for integration of desktop data into semantically enabled applications. Furthermore, we present an exemplary mapping from existing desktop data to RDF data.

4.1 Overview

Aggregating existing data and metadata from the desktop has been done by both semantic desktop implementations and by (non-semantic) desktop search engines like Beagle[1], Google Desktop[2], Copernic Desktop Search[3], Microsoft Windows Desktop Search[4] or Apple Spotlight[5]. These desktop search engines index existing data and related metadata from text files, e-

[1] http://beagle-project.org/
[2] http://desktop.google.com/
[3] http://www.copernic.com/
[4] http://www.microsoft.com/windows/desktopsearch/default.mspx
[5] http://www.apple.com/macosx/features/spotlight/

mails, media files etc. These desktop search engines allow a keyword search in data and metadata. Additionally, they allow for more advanced queries like searching for different criteria of e-mails (e.g. sender, recipient, date) files (e.g. size, modification time, type) and media (e.g. artist, bitrate).

These desktop search engines only allow retrieving desktop data. Semantic desktop applications like Gnowsis[6] and Haystack[7] provide more advanced features like referencing documents, better integration and aggregation of different documents. They try to completely hide the file system from the user and allow the user to organise data completely free, without any restrictions that the exclusive usage of file systems implies.

4.2 Related Work

Gnowsis and Haystack, two semantic desktop applications, are using existing metadata from the desktop through adapters for different data.

Gnowsis [33] uses a similar adapter approach to access metadata from desktop applications, external data sources like SQL servers, IMAP servers etc. Gnowsis distinguishes between three different adapter types: graph and query adapters, concise bound description adapters and file extractors. Graph and query adapters provide either a RDF graph representation or a query language like SPARQL or RDQL. These adapters return a list of statements that match the given query. Concise bound description adapters return a small subgraph that describe the properties of one resource, given the resource's URI. File extractors read a file, parse its content and extract some metadata. File extractors are similar to the adapter technique used in our system. Our system does not distinguish between different adapter types, they all expose an RDF graph.

In Haystack, adapters are called Observers. They *"act as transparent intermediaries between the user and external information sources"* [1]. They are implemented as proxies that observe the network communication from client applications. This reduces the number of observers as different applications for the same domain use the same network protocol. For example, only one proxy for HTTP data has to be written to observe the user behaviour in different web browsers. Additionally, a crawler traverses the user's files periodically.

4.3 Architecture

A major design goal was the transparent usage of existing metadata within ActiveRDF (see section 2.3 for details about ActiveRDF). A developer

[6] `http://www.gnowsis.org`
[7] `http://haystack.lcs.mit.edu/`

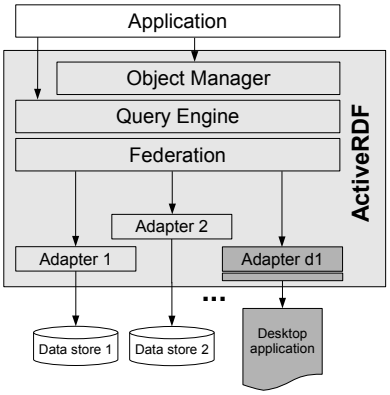


Figure 4.1: Architecture of ActiveRDF with adapters to desktop applications

should be able to access existing metadata from the desktop, which is usually not available in RDF, as if it were stored in a RDF graph. Figure 4.1 shows the architecture of ActiveRDF with additional adapters for desktop import. The figure shows that the access to desktop metadata is completely transparent for applications that use the object manager as well as for those using the query engine. With this architecture, external metadata from the desktop is not duplicated within ActiveRDF. This avoids the need for synchronisation between ActiveRDF and external data sources. In addition to adapters to triple stores, adapters that provide access to desktop data must map the mostly non-RDF data to an RDF ontology. It is possible to create a dedicated ontology for each specific application to allow an easy mapping between the application and ActiveRDF. It is also possible to use existing popular ontologies that are widely used for specific domains (e.g. the Friend-of-a-Friend[8] ontology for address book data, the SIOC[9] ontology for newsgroup and forum posts). We decided to user the latter approach to ensure a more application independent representation on the ActiveRDF side. We can change the underlying desktop application (in the same domain) without changing any application, only the different data source has to be configured.

4.4 Prototype Implementation

We implemented an adapter to the Evolution Personal Information Management application for Linux as a proof of concept. It allows the user to access the Evolution address book via ActiveRDF. It exposes the address book data over the FOAF vocabulary. An Evolution specific extension en-

[8] `http://foaf-project.org/`
[9] `http://www.sioc-project.org/`

Class	*Evolution Property*	*RDF Property*
Contact	uid (string)	(URI)
	firstname (string)	foaf:givenname, foaf:name
	lastname (string)	foaf:surname, foaf:name
	title (string)	foaf:title
	email_addresses (Hash "OTHER"->Array[string])	foaf:mbox
	email_addresses (Hash "HOME"->Array[string])	activerdf:homeMbox (rdfs:subPropertyOf foaf:mbox)
	email_addresses (Hash "WORK"->Array[string])	activerdf:workMbox (rdfs:subPropertyOf foaf:mbox)
	birthday (Time)	foaf:birthday (Literal (mm-dd))
	home_phone	activerdf:homePhone (rdfs:subPropertyOf foaf:phone)
	work_phone	activerdf:workPhone (rdfs:subPropertyOf foaf:phone)
	mobile_phone	activerdf:mobilePhone (rdfs:subPropertyOf foaf:phone)
	im_addresses (Array[ContactIMAddress])	See ContactIMAddress section
ContactIMAddress	provider (string)	foaf:OnlineChatAccount
	address, provider == 'AIM'	foaf:aimChatID
	address, provider == 'Jabber'	foaf:jabberID
	address, provider == 'Yahoo'	foaf:yahooChatID
	address, provider == 'MSN'	foaf:msnChatID
	address, provider == 'ICQ'	foaf:icqChatID
	address, provider == 'Groupwise'	not mapped

Figure 4.2: Mapping from Evolution contacts to RDF data.

sures a more complete mapping to RDF. Instances of Evolutions `Contact` class are mapped to RDF resources whose `rdfs:type` is `foaf:Person`.

Figure 4.2 shows a detailed specification of the mapping from Evolution classes and attributes to FOAF (RDFS) classes and properties. Some Evolution attributes such as `firstname`, `lastname` and `title` can be directly mapped to according FOAF properties. For other attributes like e-mail addresses or phone numbers it is necessary to create subclasses of existing foaf properties to ensure that no information from Evolution is lost. If we would map each email address directly to FOAF properties, the distinction between home and work email addresses is lost. To circumvent this, the additional properties `activerdf:homeMbox` and `activerdf:workMbox` are created. They are subproperties of `foaf:mbox`, so an application using the generated data does not need to know about these Evolution-specific properties. The address data of Evolution's `Contact` class cannot be mapped to FOAF properties, thus they are not listed here. A mapping of these attributes to other ontologies, especially the vCard ontology[10], is possible but not implemented in the present case.

The following listing shows a sample query with ActiveRDF against the Evolution address book.

```
ConnectionPool.add (:type => :evolution)
ObjectManager.construct_classes
sebastian = Person.find_by_firstname('Sebastian')
print sebastian.firstname
print sebastian.lastname
print sebastian.mbox
```

First, the Evolution adapter is initialised (line 1) and the object manager constructs the class hierarchy (line 2). Then the address book is queried for a person whose first name is "Sebastian" (line 3). Finally, we print the full name and the e-mail address of Sebastian (lines 4-6). The output of this snippet is:

```
Sebastian
Gerke
sebastian.gerke@deri.org
```

In this chapter, we presented an architecture for a seamless integration of desktop data into ActiveRDF. We compared the architecture to other semantic desktop application's desktop integration architectures. Finally, we presented an example mapping of Evolution address book entries to FOAF persons and gave an example on how to query Evolution data from within ActiveRDF.

[10] `http://www.w3.org/2006/vcard/ns`

Chapter 5

Annotation Extraction

In Semantic Wikis, users probably first enter the content in natural language text and then they possibly create additional semantic annotations based on the text they typed. To help the user creating these semantic annotations, we developed a suggestion system. This system suggests annotations based on the existing text. Then these suggestions can be confirmed and filled with the appropriate values. The suggestion system presented in this chapter is very simple to ensure a low resource consumption (memory- and time-wise). It is based on keyword matching in RDF schemas.

5.1 Related Work

Many work exists in the area of natural language processing and semantic authoring. Several approaches exist to extract semantic annotations from text. Some of them extract named entities or try to derive complete ontologies from text [22]. Other approaches use controlled languages (i.e. natural language with a simplified grammar and vocabulary) for semantic authoring [10].

5.2 Implementation

We implemented a prototype that demonstrates a basic algorithm for annotation suggestion based on natural language. Listing 5.1 shows the algorithm in detail.

First, the last sentence from the wiki text is extracted (line 2). Then, a list of keywords is extracted from the last sentence (line 3). This is done by an external part-of-speech tagger[1]. The verbs and nouns from the sentence form the keyword list. Then the keywords are concatenated with a space

[1] http://www.markwatson.com/opensource/

Listing 5.1: Method for generating suggestionsMethod for generating suggestions

```
def generate_suggestions()
  sentence = get_last_sentence()
  keywords = get_nouns_and_verbs(sentence)
  query_text = keywords.join(' ')
  if query_text.size > 0
    query = Query.new.distinct(:pred)
      .where(:pred,
             Namespace.lookup(:rdf, :type),
             Namespace.lookup(:rdf, :Property))
      .where(:pred, :keyword, query_text)
    suggestions = query.execute
  end
end
```

character (line 4). This string is then used to perform an ActiveRDF keyword search on all properties in loaded schemas (line 6 – 11). ActiveRDF searches all literals that are attributes of instances of the `rdfs:Property` class.

Figure 5.1 shows the prototype that uses the presented algorithm. It consists of three major components: A text field for entering a schema URL, a text area for the wiki text and a list that contains the suggested predicates. The user can manually request a suggestion with a button at the bottom of the program window. But suggestions are not only generated on request but also generated automatically after a certain time of typing inactivity. A double-click on a suggestion adds this predicate to the wiki text and moves the cursor behind the new predicate to allow entering the value of this predicate.

5.3 Outlook

The existing prototype provides only basic functionality to demonstrate the potential benefit of such an suggestion system. Further improvements are necessary to ensure a greater benefit from such a system. A possible improvement is the usage of a thesaurus. With a thesaurus, different words with a same or similar meaning would yield a suggestion. Another option is the restriction to the `rdfs:label` of a predicate instead of searching in all attributes of a predicate. This would yield a higher precision, because especially the `rdfs:comment` attribute can contain many words that are not synonyms for the desired attribute.

Another direction of improvement is the usage of training data to learn rules for annotation suggestion from natural language. This would require more effort to create the training sets for different ontologies manually. For each ontology, a training set has to be created. Techniques like those used

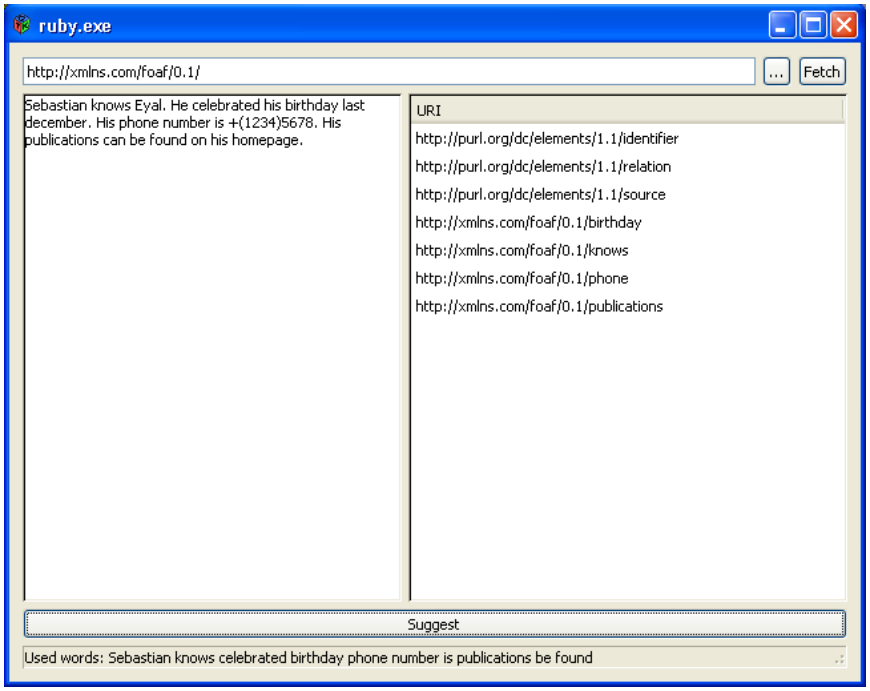


Figure 5.1: User interface of the annotation extractor prototype

in [9], [35] are options for such a machine learning approach. These are information extraction methods for free text.

Chapter 6

Collaborative Annotations

The Semantic Web is decentralised in terms of autonomy, allowing everyone to make any statement, but centralised in terms of vocabulary: others can only understand statements that use familiar terminology. Given this situation, we consider the following problem: how to ensure that individuals, free to use arbitrary terminology, converge towards shared vocabularies?

As a particular use case we consider authoring in Semantic Wikis [28, 30, 38]. These enhanced Wikis allow users to describe information both in free text and through semantic descriptions. Allowing users to make arbitrary statements is important, since it ensures domain-independence of the Wiki.

Without further considerations, the authoring freedom in Semantic Wikis would result in statements with different vocabularies, defying the purpose of the Semantic Wiki. A terminology policy could be enforced but that would highly restrict users. A suggestion mechanism, recommending terminology based on the dataset, would help converge terminology without forcing users, as demonstrated in collaborative tagging [21, 24].

In collaborative data entry, participants construct a dataset by continuously and independently adding further statements to existing data. Each participant faces the question: *when creating Semantic Web data, which vocabulary to use*? To ensure convergence, the answer is: use the most relevant and frequently occurring vocabulary.

Finding the most *frequent* vocabulary is straightforward: one can simply count the occurrences. We therefore focus on finding the *relevant* vocabulary. Datasets typically contain heterogeneous data. Finding the vocabulary that is relevant for one resource therefore means: finding similar resources and use their vocabulary.

Problem statement Our problem is thus to suggest relevant and frequent terminology for extending a resource in an RDF dataset based on similarity with other resources and our question is *how well simple algorithms solve this problem*?

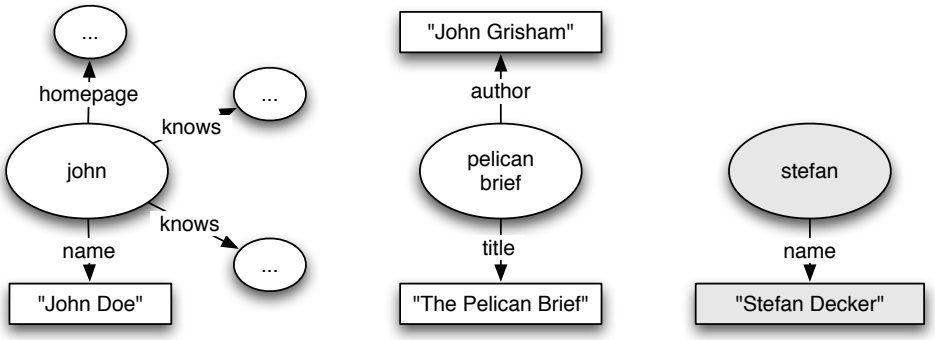


Figure 6.1: Example knowledge base

We present two algorithms that address this problem, based on the following hypotheses that simple algorithms do well enough: (a) computing resource similarity based only on *outgoing arcs* yields good results; (b) approximating resource similarity through *pairwise predicate co-occurrence* yields good results.

We will present the two algorithms in sections 6.1 and 6.2, and their implementation in section 6.3. We verify our hypotheses and the performance of these algorithms empirically in section 6.4. We conclude with a discussion of related work in section 6.5.

6.1 Classification-based algorithm

The task of the suggestion algorithm is to find, for a certain resource in focus, predicates to further describe that resource. The general idea of the classification-based algorithm is to divide the knowledge base in two groups, those similar to the current resource and those not similar, and to suggest the frequently occurring predicates from the similar group.

For example, figure 6.1 shows a simple knowledge base with three resources: the person "John", with his name, some friends, and homepage, the book "The Pelican Brief", with its title and author, and the person "Stefan", with his name. We want to suggest relevant predicates for "Stefan" based only on the given graph.

The algorithm consists of two steps, as shown in listing 6.1. In the first step, we divide all existing resources in the knowledge base into two sets, the similar and dissimilar ones. In the second step, we look at all predicates from the similar group and rank them using a ranking function. In the remainder of this section, we explore each step in more detail: how to define similarity between resources, and how to rank the selected predicates.

Listing 6.1: Classification-based algorithm

```
def suggest(r, resources)
  # select similar resources
  similar_resources =
    resources.select { |r'| similarity(r,r') > threshold }

  # then collect all predicates from similar resources
  candidates = similar_resources.collect { |r'| r'.predicates }

  # then rank all candidate predicates
  return rank(candidates)
end
```

6.1.1 Preliminaries

In addition to the definition of a RDF graph in section 2.1.2, we define some more sets:

Definition 3 (Outgoing edges) *The set of outgoing edges $E_o(v)$ of a vertex is defined as: $E_o(v) = \{e \in E | source(e) = v\} \subseteq E$. The bag of labels $L(E)$ of a set of edges is defined as $L(E) = [l(e)|e \in E]$. The bag of labels $L_o(v)$ of outgoing edges of a vertex v is defined as $L_o(v) = L(E_o(v))$. The set of outgoing edges of v whose label is l is defined as $E_o(v,l) = \{e \in E_o(v) | l(e) = l\}$.*

6.1.2 Classification step

In the first step, we classify resources into those similar to the current one, and those not similar. The main requirement for the similarity metric is domain-independence: the algorithm should not rely on domain-specific knowledge. We use two well-known, widely used generic similarity metrics [8, 11]: the *containment* of one resource in another and their mutual *resemblance*.

Since we are interested in suggesting new predicates, we use these metrics to compare existing predicates of resources. Containment thus defines resource similarity as the amount of predicates of the first resource that are also contained in the second resource, as shown in equation (6.1). Resemblance measures how many of all predicates used in at least one of the two resources are used in both resources, as shown in equation (6.2). For example, in figure 6.1, the resource "Stefan" uses the predicate "name" and the resource "John" uses "name", "knows" and "homepage", resulting in a containment value (of "Stefan" in "John") of 1 and a resemblance of $\frac{1}{3}$.

$$s_c(v', v) = \frac{|O(v) \cap O(v')|}{|O(v)|} \tag{6.1}$$

$$s_r(v', v) = \frac{|O(v) \cap O(v')|}{|O(v) \cup O(v')|}. \tag{6.2}$$

Since predicates can have multiple values, when computing this containment or resemblance metrics we need to decide whether to count multiple predicate occurrences once or several times.

In the example, the resource "John" uses the "knows" predicate twice with different values; we can either count these two occurrences only once, thus using $O(v)$ as a set, as shown in equation (6.3). The resemblance between "Stefan" and "John" would then be $\frac{1}{3}$. But we could also count each occurrences separately, using $O(v)$ as a bag as shown in equation (6.4), yielding a resemblance of $\frac{1}{4}$.

$$O_s(v) = \{l(e)|e \in E_o(v)\} \tag{6.3}$$

$$O_b(v) = [l(e)|e \in E_o(v)] \tag{6.4}$$

If we generalise from these two choices, the result of the first phase is the set of similar resources $V_s(v)$, as defined in equation (6.5), where s_t is some similarity threshold and $s(v, v')$ is either resemblance or containment measure. For example, with a threshold of 0.9 the set of similar resources to "Stefan" would consist only of the resource "John".

$$V_s(v) = \{v' \in V : s(v, v') \geq s_t\} \tag{6.5}$$

6.1.3 Ranking step

After classifying all resources into two groups we collect all predicates from the set of similar resources $V_s(v)$ and use them as candidates for the suggestion. Since there might be many candidates, we need to rank these candidates and suggest the more useful predicates first. The most straightforward ranking function is based on the occurrence frequency of these predicates in the set of similar resources.

In this example, since only the resource "John" is similar to "Stefan", the candidates would be "knows" and "homepage", ignoring the predicates that "Stefan" uses already. Out of these two candidates, "knows" would be ranked first since it appears most frequently.

But again, since predicates in RDF can be multi-valued, we can define the (relative) occurrence frequency of a label l in the set of similar resources $V_s(v)$ in two ways. We can either count each predicate occurrence, as shown in equation (6.6). Or we can count each occurrence only once, or stated differently, count the set X of resources that use l in their outgoing edges and divide them by the total number of resources, as shown in equations

(6.7). In the latter case, "knows" and "homepage" would be ranked the same since they are both used by one resource.

$$r_v(e) = f_s^p(v,l) = \frac{\sum_{v' \in V_s(v)} |E_o(v',l)| \cdot w(v,v')}{\sum_{v' \in V_s(v)} |E_o(v')| \cdot w(v,v')} \tag{6.6}$$

$$r_v(e) = f_s^r(v,l) = \frac{\sum_{v' \in X} w(v,v')}{\sum_{v' \in V_s(v)} w(v,v')}$$
$$X = \{v \in V_s(v) | l \in O(v)\} \tag{6.7}$$

In both methods of counting, we could allow for a weighting factor $w(v,v')$. The reason for this is that even in the set of similar resources $V_s(v)$, some are more similar than other: in ranking the predicates, it would be natural to "promote" the predicates from similar resources over those from less similar resources. If we choose to prefer predicates from resources more similar to v, the weight factor could be given by the resource similarity, shown in equation (6.8). A simpler approach would not to weigh the predicates, as shown in equation (6.9). In our example, these methods would yield the same ranking since both candidates originate from the same resource "John".

$$w_s(v,v') = s(v,v') \tag{6.8}$$

$$w_c(v,v') = \begin{cases} 1 & : \quad v' \in V_s(v) \\ 0 & : \quad v' \in V_n(v) \end{cases} \tag{6.9}$$

6.1.4 Qualitative results

To investigate our hypothesis, we have evaluated the performance and quality of the algorithm using various different datasets. We are interested in the quality of the basic algorithm (using containment, counting multi-valued predicates only once, and without weighting) and in whether the various parameters, while reducing simplicity, improve the basic algorithm. We present and discuss these results in section 6.4.

6.1.5 Performance

Regarding the runtime performance of the algorithm, we can analyse the description in listing 6.1. We see that, ignoring data access, the overall algorithm should run linearly to the number of resources: The first phase, classifying the similar resources, runs linear to the number of resources r and the average number of predicates per resource p: comparing the similarity of each resource against the one resource in focus by comparing all their predicates. The second phase, ranking, is linear in the number of candidates

c. The complete algorithm would therefore run in $O(r \cdot p + c)$, which is linear in r, since p will be constant on average and c is presumably smaller than r.

However, in practise we cannot ignore lookup performance on large datasets. To compute similarity, we need to lookup all predicates of each resource. Depending on the lookup performance of the used data store, this could cause the whole algorithm to run logarithmic or even quadratic to the size of the dataset, rendering the algorithm impracticable for reasonably large datasets.

A simple solution would be to materialise the similarity between resources in memory, obliterating the need for data lookup during suggestion time. Direct materialisation however has two problems: the required memory space would be quadratic in the size of the dataset, and updating one resource (prone to happen often in a data entry scenario) would require recalculation of all similarity values with respect to this resource.

The next algorithm remedies exactly this problem and allows materialisation without large memory requirements.

6.2 Co-occurrence-based algorithm

The general idea of the co-occurrence-based algorithm is to approximate resource similarity through the co-occurrence of predicates. Since usually datasets contain far less predicates than resources, predicate co-occurrence requires far less space than resource similarity. We then further reduce the required space by not considering the complete power set over all predicates, but instead approximate full co-occurrence through binary co-occurrences. We thus consider only pairwise occurrences of predicates, suggest predicate candidates for each pairwise occurrence, and combine these candidates through intersection.

We therefore make two assumptions on the probabilistic model of the dataset: (1) that predicate co-occurrence correlates with resource similarity, and (2) that considering binary predicate co-occurrences to be independent events (which they are not) yields acceptable predictions. The latter allows us to pairwise consider binary co-occurrences instead of all permutations.

The algorithm is based on association rule mining [2, 36] used for recommendations in e.g. online stores: when buying one book, other books that are often bought together with this book are recommended. In our case, books are replaced by predicates and shopping transactions by resources.

6.2.1 Precomputation step

To better show the details of the algorithm, we extend our earlier example, adding the person “Sebastian” and some more statements about John, as shown in figure 6.2. Again, we want to suggest further predicates to the resource “Stefan”.

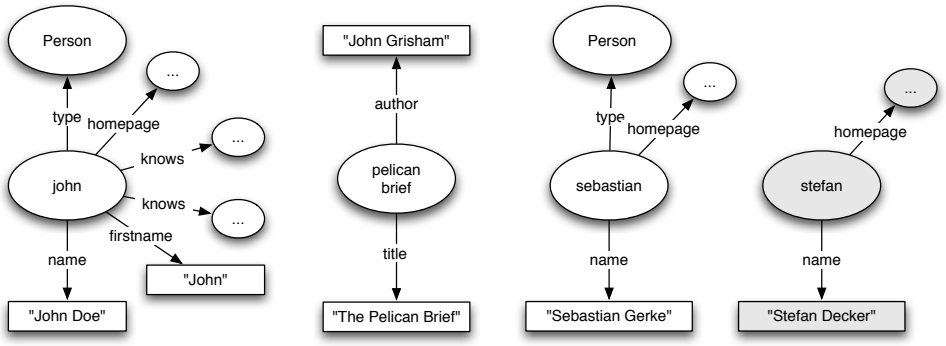


Figure 6.2: Extended Knowledge Base

predicate	*freq.*
type	3
name	2
knows	1
homepage	2
firstname	1
author	1

(a) occurrence

	type	name	knows	homepage	firstname	author
type	3	2	1	2	1	1
name	2	2	1	2	1	0
knows	1	1	1	1	1	0
homepage	2	2	1	2	1	0
firstname	1	1	1	1	1	0
author	1	0	0	0	0	0

(b) co-occurrence

Table 6.1: Predicate occurrence and co-occurrence frequency

In the first step we calculate usage statistics of predicates in the knowledge base. We count for each predicate, the resources that use this predicate, defined in equation (6.10). Secondly, we count for each pair of predicates, the number of times they co-occur together in the same resource, as defined in equation (6.11). The particular statistics for the example in figure 6.2 are given in table 6.1a and table 6.1b.

$$occ(p) = |\{v \in V | p \in L_o(v)\}| \tag{6.10}$$

$$coocc(p_1, p_2) = |\{v \in V | p_1 \in L_o(v) \wedge p_2 \in L_o(v)\}| \tag{6.11}$$

6.2.2 Suggestion step

In the second step, we compute suggestions for a given resource. We consider all predicates in the knowledge base that occur more than once with each of

candidate	*name*	*homepage*	*confidence*
type	1.0	1.0	1.0
knows	0.5	0.5	0.25
firstname	0.5	0.5	0.25

Table 6.2: Relative co-occurrence ratios for Stefan

the predicates from "Stefan" as suggestion candidates, as defined in equation (6.12). In our example, the predicates "type", "knows", and "firstname" are candidates for the resource "Stefan".

$$cooccurring(p_1) = \{p_2 : coocc(p_1, p_2) > 1\} \tag{6.12}$$

For each candidate we calculate our confidence in suggesting it. As shown in equation (6.13), the confidence for suggesting a predicate p for a selected resource r, is formed by combining the confidence for p from each of r's predicates p_i. In the earlier example, the total confidence for suggesting "type" is computed by combining $confidence(name \Rightarrow type)$ and $confidence(homepage \Rightarrow type)$.

$$confidence(p, r) = \prod_{p_i \in cooccurring(p) \cap L_o(r)} confidence(p_i \Rightarrow p) \tag{6.13}$$

Each constituent is computed as shown in equation (6.14): the confidence for suggesting any p_2 based on the existence of a p_1 is given as the co-occurrence frequency of p_1 and p_2 relative to the occurrence frequency of p_1 by itself. In our example, p_2, the candidate, would be "type", "knows", or "firstname", and p_1, the existing predicates, would be "name" and "homepage". Intuitively, we consider a relatively frequent co-occurrence as evidence for predicting p_2.

$$confidence(p_1 \Rightarrow p_2) = \frac{coocc(p_1, p_2)}{occ(p_1)} \tag{6.14}$$

In our example, as shown in table 6.2, "type" co-occurs with both predicates of "Stefan" 100% of the time, whereas the two other candidates ("knows" and "firstname") co-occur only 50% of the time with each of the predicates of "Stefan". We rank each candidate by the combined (unweighted) confidence: in this example, "type" will be ranked first, with a combined confidence of 100%, and the other two second, with a combined confidence of 25%.

Listing 6.2: Co-occurrrence as database views

```
create view occurrence as
 select p, count(distinct s) as count
 from triple
 group by p;

create view cooccurrence as
 select t0.p as p1, t1.p as p2, count(distinct t0.s) as count
 from triple as t0 join triple as t1 on t0.s = t1.s and t0.p != t1.p
 group by t0.p, t1.p
```

6.3 Implementation

We have implemented both algorithms in Ruby. We use the ActiveRDF [29] data store abstraction layer which allows us to run this algorithm on various RDF data stores. The implementations are distributed as part of the ActiveRDF. We have also implemented the co-occurrence algorithm as a wrapper for an RDF data store, in particular for the rdflite[1] RDF store.

Since rdflite uses a relational database with one table, `triple(s,p,o)`, we have implemented the (co)occurrence statistics as views on this database, comparable to [18]. Depending on the relational database, these views can be materialised or computed for each suggestion. The views, shown in listing 6.2, are a straightforward translation of the equations (6.10) and (6.11) given before.

6.3.1 Example suggestions

Figure 6.3 shows an example of our suggestion system, for a randomly chosen resource from a dataset[2] about the Mindswap research group. The resource (a blank node representing Dan Connolly) and its predicates, such as name and email address, are listed on the left-hand side. Our suggestions, based on the other resources in this dataset, are listed in ranked order on the right-hand side.

6.4 Evaluation

A predicate suggestion system is a kind of recommender system, using the opinions of a community to help individuals decide between a potentially overwhelming set of choices [16, 32]. In our case, this "potentially overwhelming set of choices" is formed by the terminology (ontologies or schemas) available.

[1] `http://wiki.activerdf.org/rdflite/`
[2] `http://www.cs.umd.edu/~hendler/2003/MindPeople4-30.rdf`

```
_:#1 foaf:name Dan Connolly .
_:#1 owlweb:name Dan Connolly .
_:#1 foaf:mbox mailto:connolly@w3.org .
_:#1 owlweb:email mailto:connolly@w3.org .
_:#1 owlweb:homepage
          http://owl.mindswap.org/~danC/ .
_:#1 rdf:type owlweb:FamilyFriend .
```

```
1. foaf:workInfoHomepage
2. foaf:homepage
3. owlweb:personalHomepage
4. owlweb:bachelorsFrom
5. owlweb:mastersFrom
6. owlweb:homepage
7. foaf:nick
8. owlweb:phdFrom
```

Figure 6.3: Suggested predicates (bottom) for example resource (top)

Evaluations of recommender systems can be divided into two categories [14, 16]: when regarding recommendations as an information retrieval problem (selecting the interesting predicates from all possible predicates), evaluation is usually performed off-line, focused on accuracy, and measured using precision and recall. When, on the other hand, recommendation is approached as a machine learning regression problem (learning and predicting user's annotation preferences), evaluation is commonly performed online, focused on utility and usefulness, and measured using a training set and a test set.

6.4.1 Evaluation approach

Our evaluation combines both the information-retrieval and the machine-learning approach: we show both precision and recall ratings and evaluate our approach using training/testing datasets through a commonly applied technique of evaluating prediction of deleted values from existing data [16].

Because the distribution of data can alter the performance of the algorithms quite severely, we evaluated on five existing RDF datasets: a webcrawl[3] of arbitrary RDF, the Mindswap research group[4], a FOAF dataset[5], a terror dataset[6] augmented with terrorist data, and the ontoworld.org Semantic Wiki[7]. These datasets have differing characteristics, as shown in Table 6.3: both large and small, with homogeneous and heterogeneous data, and both highly structured and highly unstructured distribution. Figure 6.4

[3] http://www.activerdf.org/webcrawl_10k.nt
[4] http://www.cs.umd.edu/~hendler/2003/MindPeople4-30.rdf
[5] http://rdfweb.org/2003/02/28/cwm-crawler-output.rdf
[6] http://reliant.teknowledge.com/DAML/TerroristActs.owl
[7] http://ontoworld.org/RDF/

dataset	*classes*	*resources*	*triples*
webcrawl	2	112	6766
mindpeople	14	273	1081
foaf	4	3123	10020
terror	25	1553	16632
ontoworld	42	4467	28593

Table 6.3: Evaluation datasets

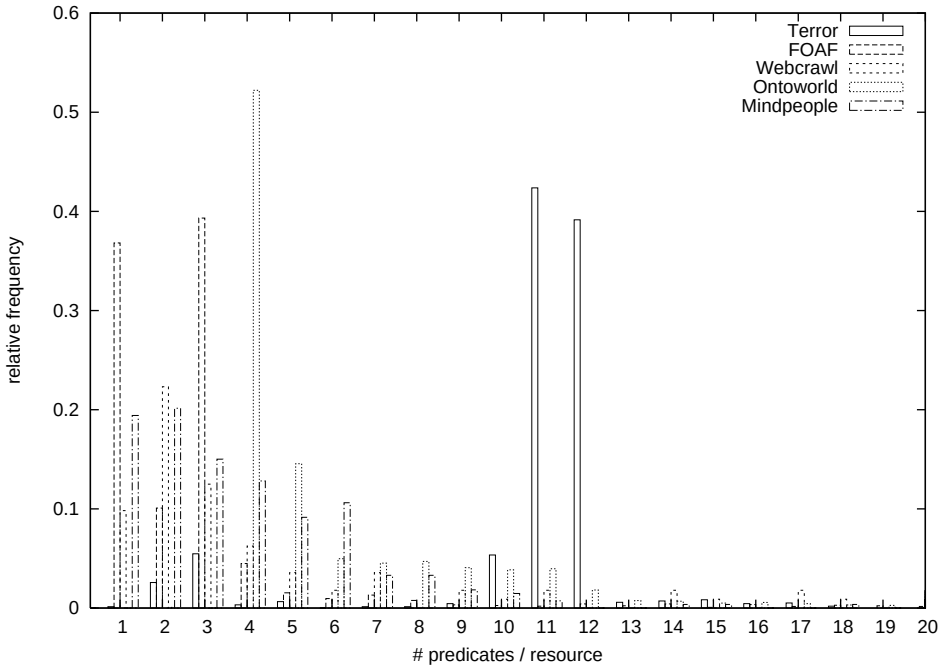


Figure 6.4: Distribution of the number of predicates per resource.

shows the relative distribution of the number of predicates per resource. It shows that resources from the terror dataset in almost all cases have eleven or twelve predicates. Resources from other datasets have usually far less predicates per resource: Those from the ontoworld dataset have mainly four or five predicates, whilst resources from the webcrawl dataset usually have up to three, those from the mindpeople dataset usually up to six predicates. Resources from the FOAF dataset have one or three predicates. All datasets besides the terror dataset have very few resources that have more than 20 predicates. This is not printed in the plot to preserve readability. Table A shows all absolute frequencies of predicates per resource.

Our primary evaluation technique is prediction of deleted values: we pick a random resource from the dataset as a candidate for which further predicates should be suggested. We then randomly remove one ore more

statements about this candidate and analyse if and at which rank position the removed predicates are re-suggested. Repeated over n random resources this yields the *average re-suggestion rate* (how often was the deleted predicate re-suggested), the *empty suggestion rate* (how often were no suggestions given), and the *average rank* of the re-suggested predicate. Since in practise not all suggestions can be displayed or will be considered by the user, we also show how many of the predicates were re-suggested within the top-k of suggestions.

Secondly, we measure suggestion precision (how many suggestions are valid) and recall (how many valid suggestions have we missed) based on the schema definition: we define "valid" predicates as those predicates that, according to the schema, fall within the domain of the selected candidate. For recall computation, we consider only predicates that are actually used in the dataset; since the algorithm considers only instance data, unused predicates are unattainable.

6.4.2 Results

All tests were run on an AMD Opteron 1993MHz machine with 2GB of RAM. The similarity algorithm was run 300 times over five random samples (n=100, n=150, n=200, n=250, n=300) since its performance prevented us from using the full datasets; the co-occurrence algorithm was run 20.000 times over the complete datasets. In each run, we randomly selected a resource and deleted between one and ten of its existing predicates. We then let the algorithms suggest additional predicates and compare these to the randomly deleted predicates.

We first show the results of the two primary algorithms for each dataset: table 6.4 shows the results of the classification-based algorithm, table 6.5 the results of the co-occurrence-based algorithm. The tables show, for each dataset and for all datasets combined, the re-suggestion rate, empty suggestion rate and average rank. It also shows the re-suggestion rate when only considering the top-k results, and the precision, recall, and the F_1-measure for each algorithm.

We can see that in general the co-occurrence performs better than any of the classification-based variants, especially when looking at the top-5 results. We can see that the co-occurrence algorithm has very high precision (100% on average). The co-occurrence algorithm has a slightly lower recall than the classification-based ones, due to the intersection of candidates which results in only high-confidence candidates. The F_1-measure (harmonic mean of precision and recall) shows that co-occurrence has the highest quality over all datasets.

Table 6.6 shows (again) the results for the primary algorithms and then lists the results for each classification variant, averaged over all five datasets. We see that using resemblance instead of containment yields very low re-

dataset	*resugg.*	*empty*	*rank*	*top-5*	*top-10*	*top-20*
webcrawl	0.95	0.04	1.06	0.94	0.94	0.94
mindpeople	0.80	0.19	1.30	0.79	0.80	0.80
foaf	0.92	0.06	1.30	0.92	0.93	0.93
terror	0.98	0.02	1.10	0.97	0.97	0.98
ontoworld	0.85	0.13	1.39	0.84	0.84	0.85
average	0.90	0.08	1.22	0.89	0.90	0.90

dataset	*prec.*	*recall*	F_1
webcrawl	0.96	0.73	0.83
mindpeople	0.81	0.83	0.83
foaf	0.94	0.80	0.87
terror	0.98	0.91	0.95
ontoworld 0.87	0.72	0.79	
average	0.92	0.80	0.85

Table 6.4: Results per dataset for classification-based algorithm

dataset	*resugg.*	*empty*	*rank*	*top-5*	*top-10*	*top-20*
webcrawl	1.00	0.00	1.18	0.99	0.99	1.00
mindpeople	1.00	0.00	1.23	1.00	1.00	1.00
foaf	1.00	0.00	1.51	0.95	1.00	1.00
terror	1.00	0.00	1.15	0.98	1.00	1.00
ontoworld	1.00	0.00	1.14	0.98	1.00	1.00
average	1.00	0.00	1.24	0.98	1.00	1.00

dataset	*prec.*	*recall*	F_1
webcrawl	1.00	0.74	0.85
mindpeople	1.00	0.76	0.87
foaf	1.00	0.59	0.74
terror	1.00	0.95	0.97
ontoworld	1.00	0.78	0.88
average	1.00	0.77	0.87

Table 6.5: Results per dataset for co-occurrence-based algorithm

algorithm	*resugg.*	*empty*	*rank*	*top-5*	*top-10*	*top-20*
co-occurrence	1.00	0.00	1.24	0.98	1.00	1.00
similarity (default)	0.90	0.08	1.22	0.89	0.90	0.90
resemblance (s_r)	0.10	0.86	1.01	0.11	0.11	0.11
similarity weigh (w_s)	0.90	0.09	1.24	0.89	0.90	0.90
count predicates (f_s^p)	0.91	0.08	1.48	0.89	0.90	0.91
threshold (s_t=0.8)	0.93	0.06	1.29	0.91	0.92	0.93

algorithm	*prec.*	*recall*	F_1
co-occurrence	1.00	0.77	0.87
similarity (default)	0.92	0.80	0.85
resemblance (s_r)	0.14	0.99	0.24
similarity weigh (w_s)	0.91	0.80	0.85
count predicates (f_s^p)	0.92	0.80	0.85
threshold (s_t=0.8)	0.94	0.79	0.86

Table 6.6: Results of algorithm variants (averaged over all datasets)

sults, which is most probably due to a too high threshold value. The other variations do not seem to affect the results much.

Figure B.1 shows the F_1-measure as a function of the number of predicates per resource for each dataset separately. It reveals that more predicates per resource improve the suggestion quality of the co-occurrence-based algorithm in terms of F_1-measure. But even for few predicates the algorithm performs reasonable well. The performance varies @@merklich between different datasets: while the F_1-measure for most datasets is always above 0.8, it is almost always below 0.8 for the FOAF dataset. These variations also show up in figure B.2 and B.3, which show the average rank and the top-k recall for different number of predicates per resource (for the co-occurrence-based algorithm). These plots not only show *if* a predicate is re-suggested but also at which position it is suggested. Figure B.3 reveals that almost all (more than 93 %) removed predicates appear in the first ten suggestions.

Figures B.4, B.5 and B.6 show F_1-measure, top-k re-suggestion rate and average rank as functions of the sampling size for the similarity-based algorithm. We used containment as the similarity metric (as defined in equation (6.1)), no weighting of resources (defined in equation (6.9)) and we count resources instead of predicates (equation (6.3)) for this plots. They reveal that for the F_1-measure, a sample size of 100 resources seems to be sufficient. For all datasets, the F_1-measure is nearly constant. The average rank and the top-k re-suggestions, who give a more precise insight about the suggestor, show that for all datasets, the performance does not improve significantly when using more than 250 resources. Only the top-k re-suggestion rate for the webcrawl dataset indicate a visible improvement when using 300

algorithm	*n=100*	*n=150*	*n=200*	*n=250*	*n=300*
sim. (rdflite)	1.64s	4.02s	8.51s	15.10s	30.22s
sim. (Sesame)	0.71s	1.40s	2.74s	4.33s	7.88s
co-occ. (view)	0.63s	0.78s	1.46s	1.00s	0.93s
co-occ. (constr.)	0.21s	0.27s	0.46s	0.47s	0.70s
co-occ. (query)	0.01s	0.01s	0.01s	0.01s	0.01s

algorithm	*n=1555*	*n=3123*	*n=4467*
sim. (rdflite)	–	–	–
sim. (Sesame)	–	–	–
co-occ. (view)	7.72s	9.65s	10.10s
co-occ. (constr.)	2.73s	4.71s	6.34s
co-occ. (query)	0.01s	0.01s	0.01s

Table 6.7: Runtime performance with n resources

instead of 250 resources.

Finally, Table 6.7 shows the performance times for the algorithms (only one classification variant is shown since runtime is similar for all). Figure 6.5 shows two graphs for these results; the left graph is zoomed for up to 300 resources, the right graph shows the full results.

Timing for the co-occurrence algorithm is divided in matrix construction and query answering. We evaluated the classification on two different data stores, rdflite and Sesame[8], to evaluate scaling independent of a particular data store implementation. We can see that the classification algorithm scales quadratic, which is due to the linear lookup times of the used data stores, although the Sesame data store performs much better than rdflite.

Both variants of the co-occurrence algorithm perform well and scale linearly. The materialised co-occurrence implementation performs better than the view-based, which is due to the fact that the sqlite database does not support view materialisation; as mentioned earlier, both approaches have their advantages.

The classification algorithm was too slow to include tests with more than 300 resources but that was again due to data lookups on the underlying data store: the algorithms themselves scale linearly when ignoring data-access. The materialised co-occurrence implementation shows that we can circumvent data access, leading to very good performance, without requiring large memory space.

[8] `http://www.openrdf.org`

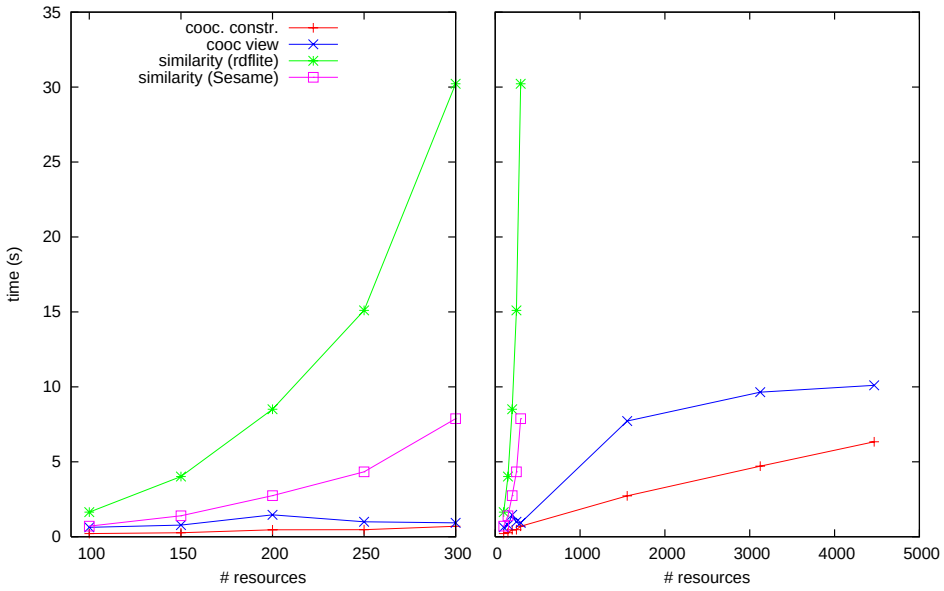


Figure 6.5: Runtime performance

6.5 Related Work

Annotation tools such as OntoMat [13] support semi-automatic annotation of documents; they suggest semantic annotation based on natural language analysis of the annotated resources, but do not take existing semantic descriptions into account. Annotea [19] supports collaborative annotations but annotations are made manually without a suggestion mechanism. Semantic Wikis such as SemperWiki [30] or Semantic MediaWiki [38] allow arbitrary Semantic Web authoring but do not guide users in selecting appropriate terminology.

Automatic schema mapping techniques [31] consider a similar problem (automatically finding relations between elements of a schema) but typically operate on class-level as opposed to instance level and use e.g. concept correlation to unify schema elements [26] whereas we try to discover combined usage patterns of predicates.

Our co-occurrence algorithm is based on association rule mining [2] but our techniques for memory conservation differ: [2] focus on advanced pruning techniques, whereas we approximate n-ary interdependencies using pairwise binary relations (resulting in a much simplified implementation). Furthermore, our technique allows online processing with incremental updates, whereas their algorithms are iterative and need to run over the complete database.

6.6 Conclusion

We have discussed the problem of choosing vocabulary during Semantic Web data entry; a crucial bottleneck, since only through shared vocabularies can meaning be established. We introduced two algorithms for suggesting possible predicates based on statistical data analysis.

The first algorithm is based on a simple intuitive principle of resource classification: we suggest predicates from similar resources. We have discussed parametric variations that differ in the definition of similarity. We showed that the quality is good ($F_1 : 85\%$) and that variations in similarity computation do not lead to much better results.

The second algorithm approximates resource similarity through pairwise predicate co-occurrence, treating predicate occurrences as independent events (which they are not). This simplifies computation and allows for memory-efficient materialisation, while still resulting in high-quality suggestions ($F_1 : 87\%$). Runtime performance of the co-occurrence algorithm is good, scales linearly with the size of the dataset, and is constant in the presence of materialisation.

We conclude that suggesting predicates based on resource similarity works well and that, for this task, similarity based on outgoing arcs seems a "good-enough" metric. Seeing that co-occurrence suggestion quality is even better than in the classification algorithm, our second hypothesis on similarity approximation using predicate co-occurrence seems to hold as well.

Chapter 7

Conclusion

We presented different techniques for simplifying the creation of semantic annotations in text, with a focus on semantic wikis. They address different facets of of wiki authoring in general as well as semantic authoring.

The first approach tackles the problem of heterogeneous wiki markup languages. It provides an architecture for storing wiki content in an markup independent manner so that each user of a wiki can use his favourite wiki markup syntax without having to adopt to wiki specific syntaxes.

The next approach is the integration of existing metadata that already exists on the user's desktop environment. It allows reuse of metadata from Word-, Excel and PDF documents, from e-mail and personal information management applications. Links between contacts in a personal information management application, e-mails and spreadsheets can be created in a semantic desktop and allow a more natural task-oriented view on data than a current application-oriented view.

Generating suggestions from natural language can help users to identify possible annotations. The approach presented in this thesis demonstrates the potential advantage that such a suggestion system might have, although the results of the presented algorithm is not very meaningful. More advanced natural language processing techniques should yield better results.

Generating suggestions that are based on existing annotations is useful mainly in collaborative environments where users can benefit from the knowledge of other users. The two presented algorithms perform quite well ($F_1 = 0.85$) in terms of quality, and the co-occurrence based algorithm also in terms of time performance, where suggestions are generated in about 0.01 seconds. Given that results, the algorithm can be useful to suggest annotations in semantic authoring environments, especially in those that are not dedicated to a certain domain.

Appendix A

Tables

#pred./res.	freq.
1	53
2	55
3	41
4	35
5	25
6	29
7	9
8	9
9	5
10	4
11	2
14	1
15	1
18	1
20	1
21	1
26	1

(a) mindpeople

#pred./res.	freq.
1	11
2	25
3	14
4	7
5	4
6	2
7	4
9	2
10	1
11	2
14	2
15	1
17	2
18	1
21	2
22	1
24	1
28	1
30	1
32	1
36	1
37	2
38	2
44	1
49	1
50	1
74	1
81	1
85	1
88	2
90	2
106	1
114	1
115	1
185	1
230	1
231	1
267	1
303	1
942	2
943	1
949	1

(b) webcrawl

#pred./res.	freq.
2	3
4	2332
5	650
6	222
7	202
8	210
9	182
10	172
11	177
12	81
13	34
14	31
15	22
16	25
17	20
18	14
19	12
20	4
21	7
22	2
23	10
24	5
25	6
26	7
27	2
28	5
31	1
32	1
33	1
35	1
37	1
38	2
39	1
41	1
42	2
43	1
45	1
46	1
47	1
48	1
51	1
52	1
54	1
55	1
56	2
57	2
59	1
60	1
85	1
96	1
154	1
209	1

(c) ontoworld

#pred./res.	freq.
1	2
2	40
3	85
4	5
5	10
7	2
8	2
9	7
10	83
11	658
12	608
13	9
14	11
15	13
16	7
17	8
18	3

(d) terror

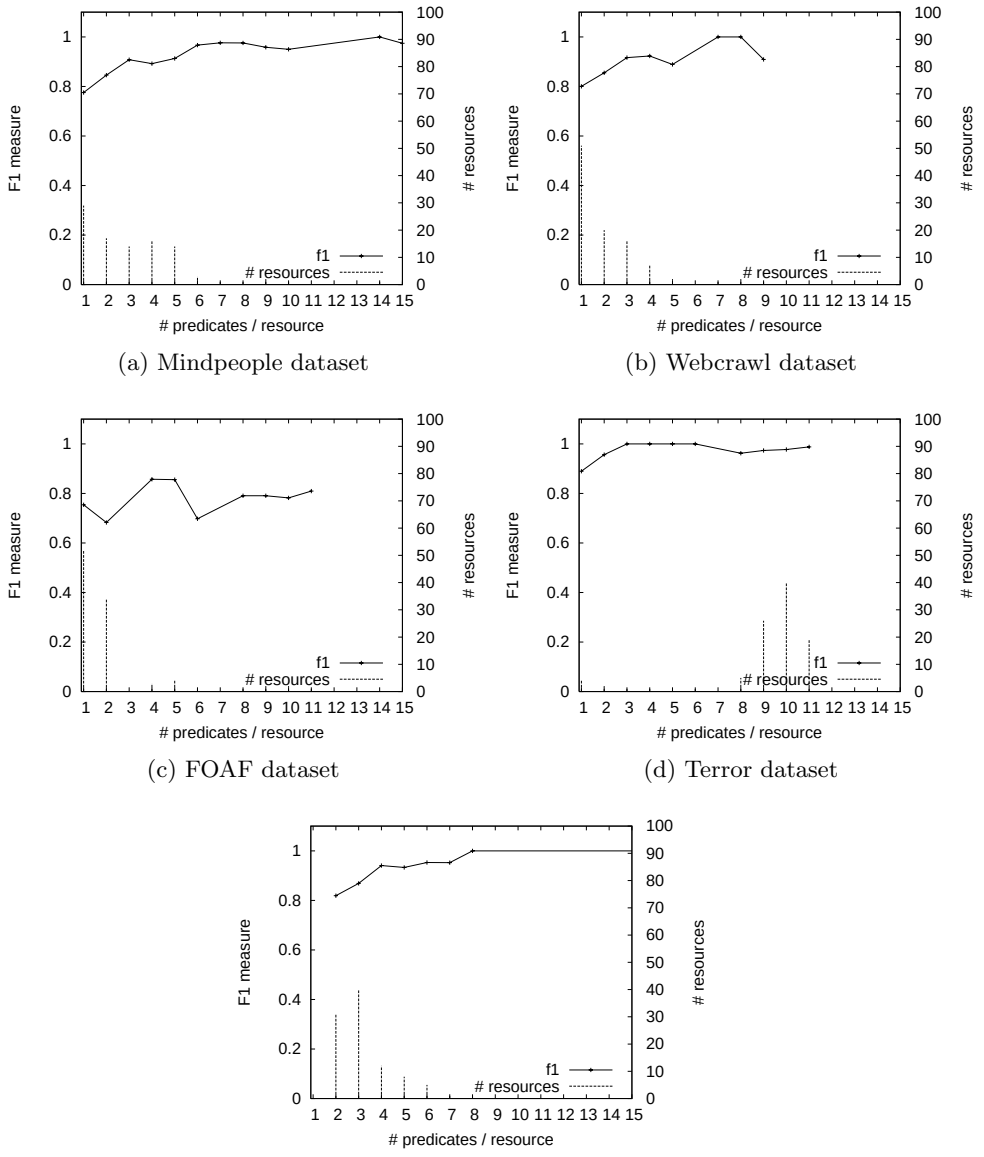
#pred./res.	freq.
1	1150
2	315
3	1228
4	140
5	48
6	30
7	41
8	24
9	10
10	8
11	6
12	13
13	7
14	13
15	2
16	11
17	5
18	9
19	7
20	5
21	4
22	3
23	5
24	1
25	5
26	1
27	4
28	4
29	1
30	1
31	1
34	3
37	2
38	2
39	1
40	3
41	1
42	1
44	4
48	1
77	1
86	1
115	1

(e) FOAF

Table A.1: Number of predicates per resource for each dataset.

Appendix B

Figures



(a) Mindpeople dataset

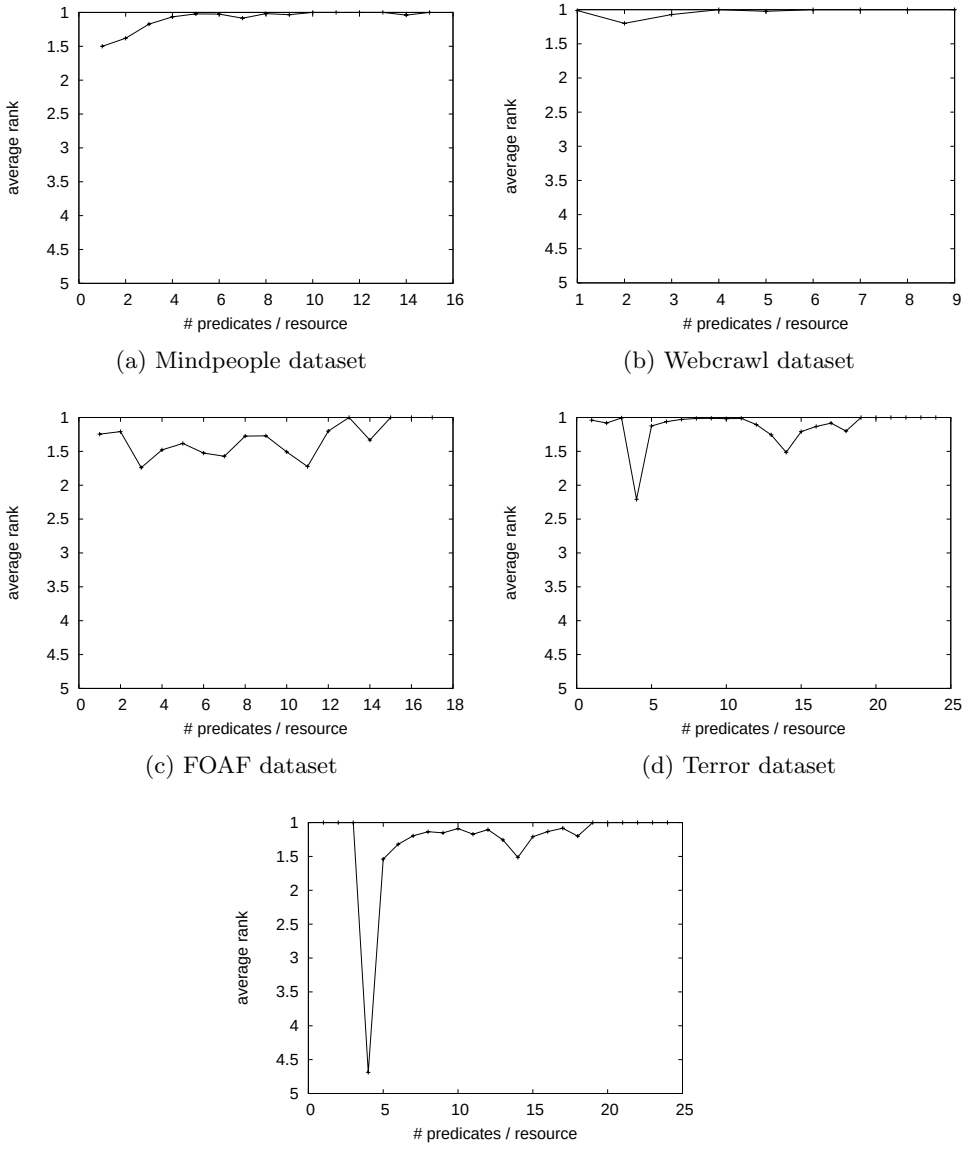
(b) Webcrawl dataset

(c) FOAF dataset

(d) Terror dataset

(e) Ontoworld dataset

Figure B.1: F1 measure as a function of the number of predicates per resource for the co-occurrence based approach. The histogram shows the number of resources with the specified number of predicates.



(a) Mindpeople dataset

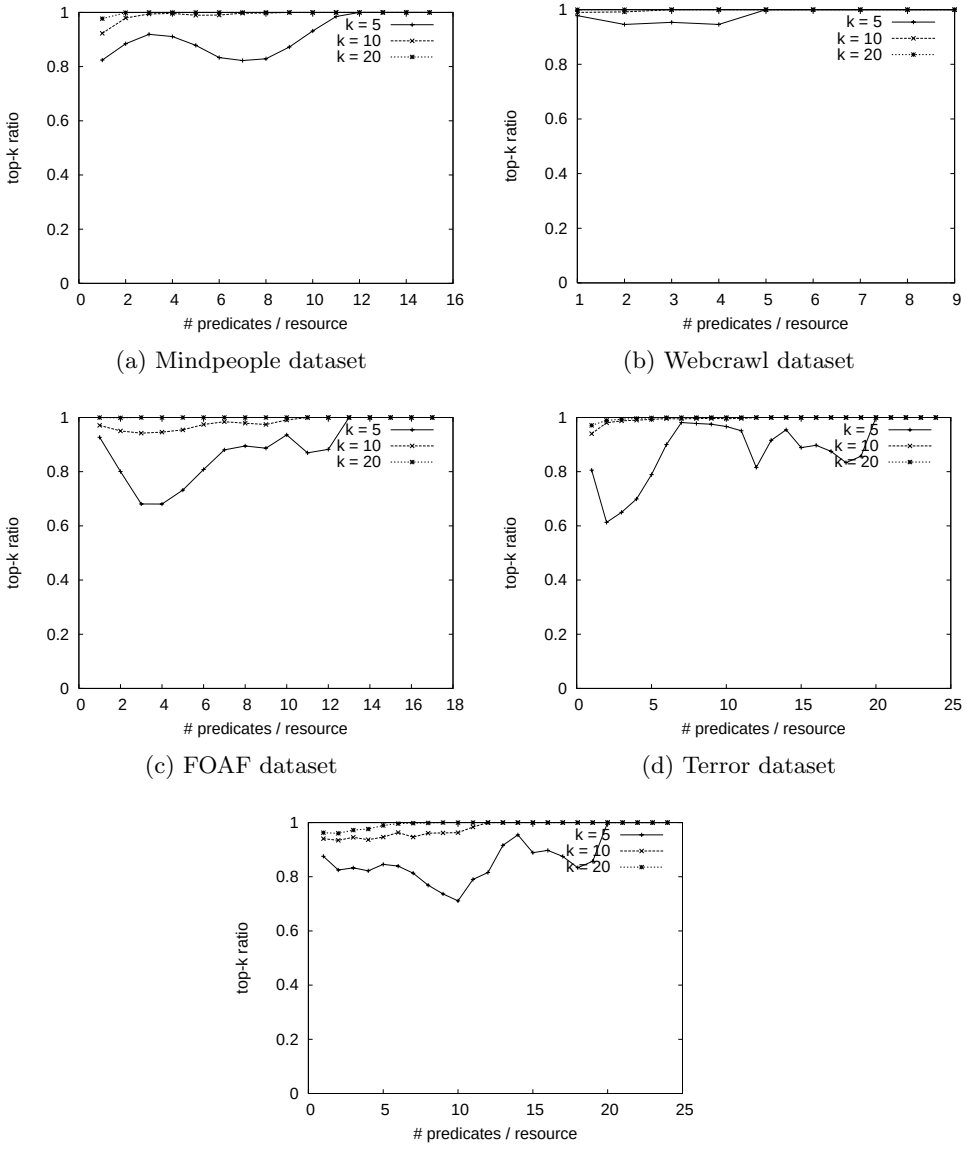
(b) Webcrawl dataset

(c) FOAF dataset

(d) Terror dataset

(e) Ontoworld dataset

Figure B.2: Average rank as a function of the number of predicates per resource for the co-occurrence based approach.



(a) Mindpeople dataset

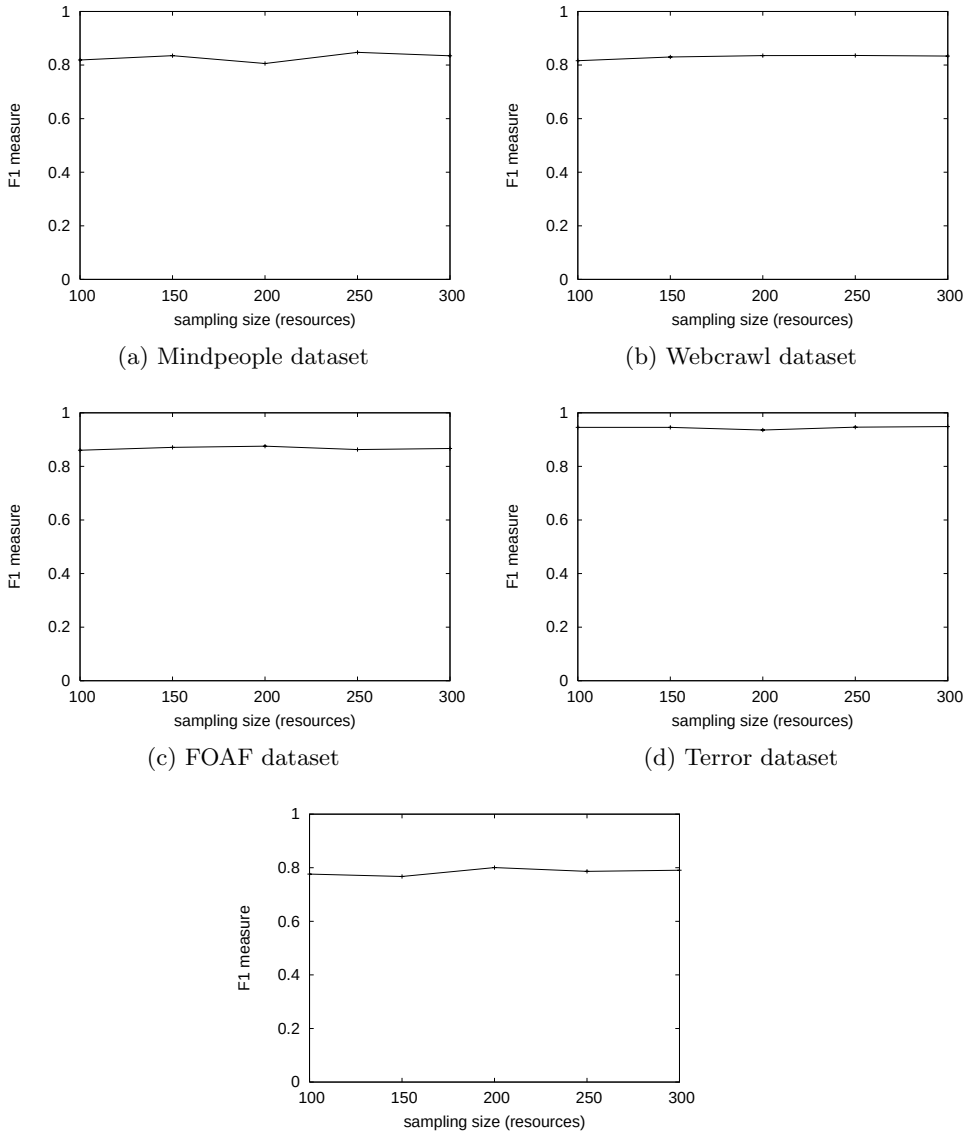
(b) Webcrawl dataset

(c) FOAF dataset

(d) Terror dataset

(e) Ontoworld dataset

Figure B.3: Ratio of resuggested predicates in top 5, 10 and 20 suggestions respectively for the co-occurrence based approach.



(a) Mindpeople dataset

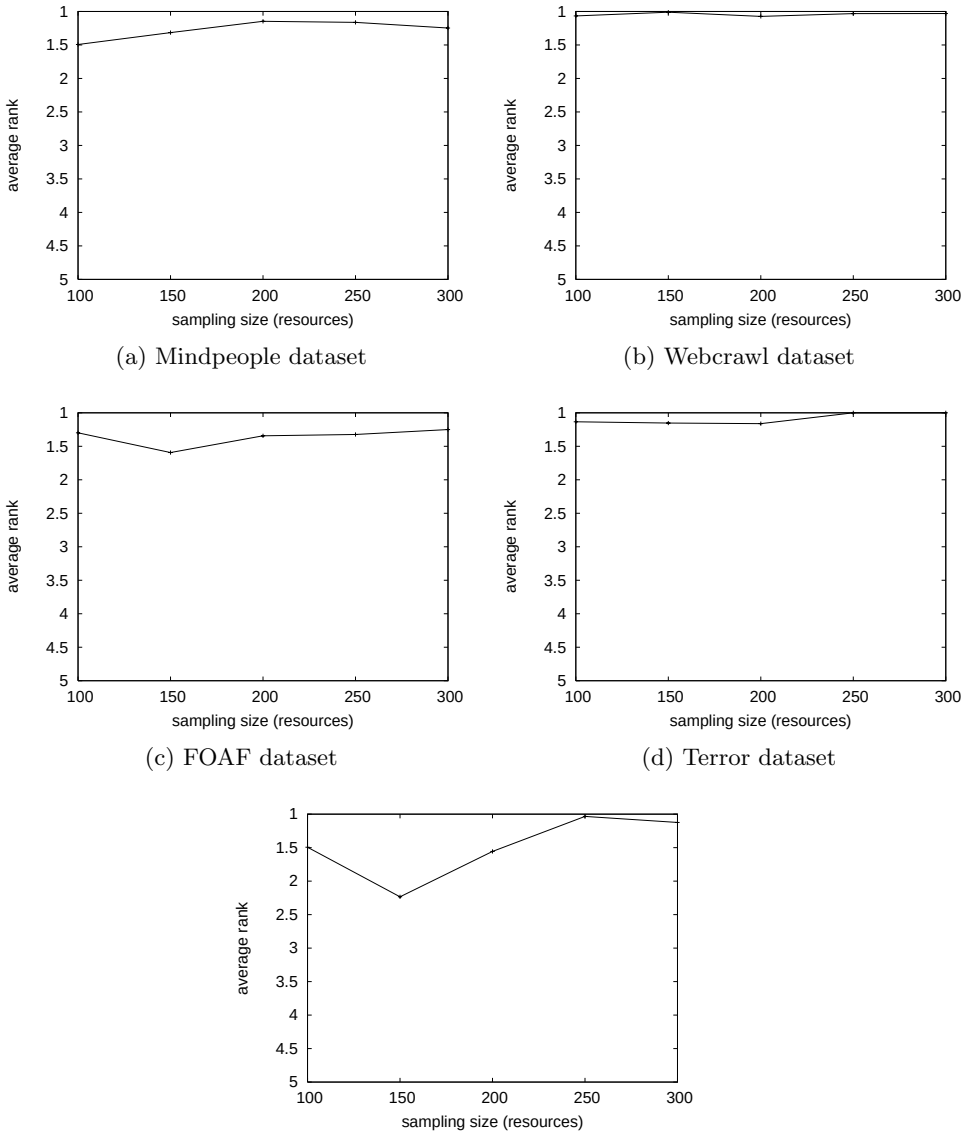
(b) Webcrawl dataset

(c) FOAF dataset

(d) Terror dataset

(e) Ontoworld dataset

Figure B.4: F1 measure as a function of the sampling size (i.e. number of random resources) resource for the similarity-based approach.



(a) Mindpeople dataset

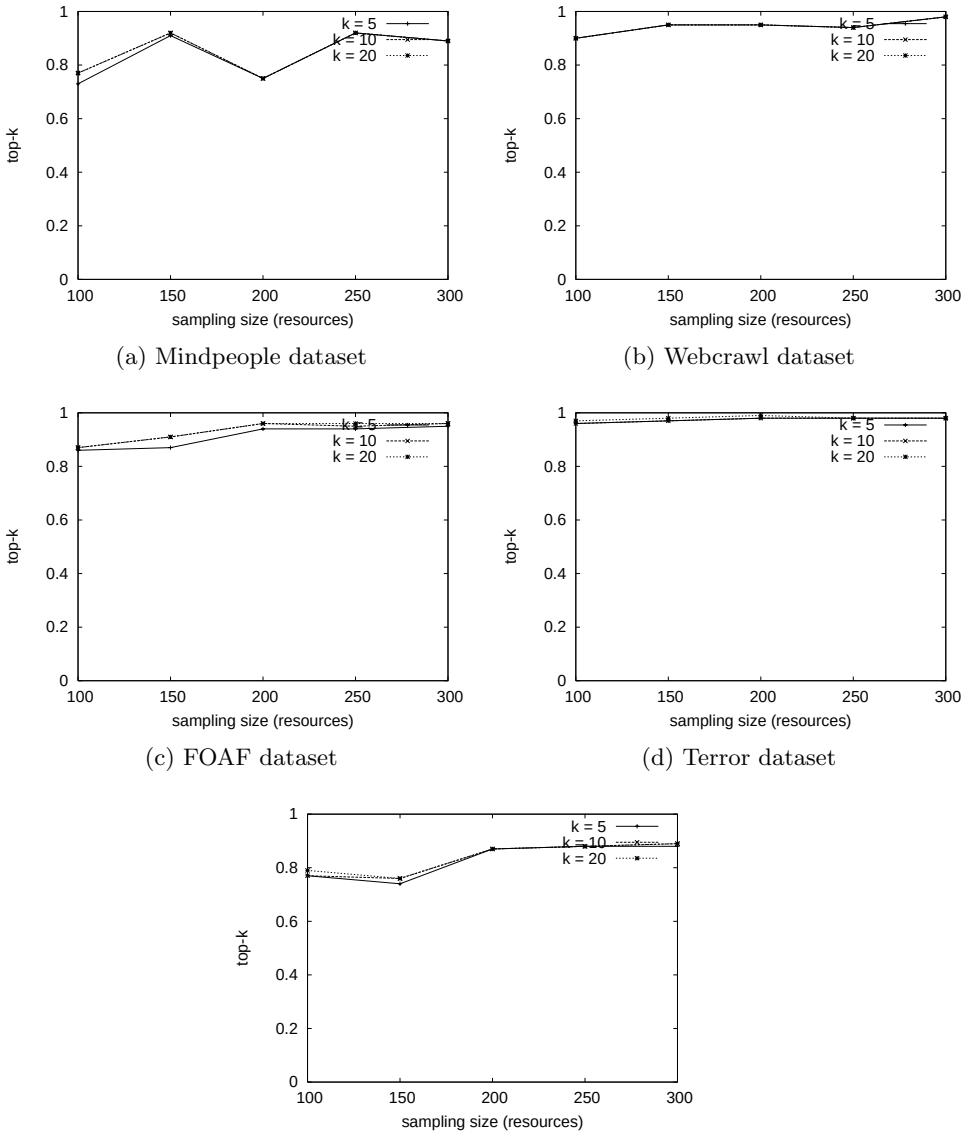
(b) Webcrawl dataset

(c) FOAF dataset

(d) Terror dataset

(e) Ontoworld dataset

Figure B.5: Average rank as a function of the sampling size for the similarity-based approach.



(a) Mindpeople dataset

(b) Webcrawl dataset

(c) FOAF dataset

(d) Terror dataset

(e) Ontoworld dataset

Figure B.6: Ratio of resuggested predicates in top 5, 10 and 20 suggestions respectively for the similarity based approach.

List of Figures

List of Tables

Bibliography

[1] E. Adar, D. Kargar, and L. A. Stein. Haystack: per-user information environments. In *CIKM '99: Proceedings of the eighth international conference on Information and knowledge management*, pp. 413–422. ACM Press, New York, NY, USA, 1999.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pp. 207–216. ACM Press, New York, NY, USA, 1993.

[3] J. Aycock and R. N. Horspool. Schrödinger's token. *Software - Practice and Experience*, 31:803–814, 2001.

[4] D. Beckett. Turtle - Terse RDF Triple Language. `http://www.dajobe.org/2004/01/turtle/`, 2004.

[5] T. Berners-Lee. Notation 3 - a readable language for data on the web. `http://www.w3.org/DesignIssues/Notation3.html`, 1998.

[6] T. Berners-Lee and M. Fischetti. *Weaving the Web -The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. Harper San Francisco, 1999.

[7] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation, W3C, February 2004.

[8] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pp. 1157–1166. Elsevier Science Publishers Ltd., Essex, UK, 1997.

[9] F. Ciravegna. (LP)2, an adaptive algorithm for information extraction from web-related texts. In *IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*. 2001.

[10] B. Davis, S. Handschuh, H. Cunningham, and V. Tablan. Further use of controlled natural language for semantic annotation. In *Proceedings of*

the 1st International Workshop on Applications and Business Aspects of the Semantic Web (SEBIZ 2006). 2006.

[11] D. Dhyani, W. K. Ng, and S. S. Bhowmick. A survey of web metrics. *ACM Comput. Surv.*, 34(4):469–503, 2002.

[12] D. Fensel, H. Lausen, A. Polleres, M. Stollberg, *et al. Enabling Semantic Web Services*. Springer, Berlin, October 2006.

[13] S. Handschuh. *Creating Ontology-based Metadata by Annotation for the Semantic Web*. Ph.D. thesis, University of Karlsruhe, 2005.

[14] C. Hayes, P. Massa, P. Avesani, and P. Cunningham. An on-line evaluation framework for recommender systems. In *Workshop on Personalization and Recommendation in E-Commerce*. 2002.

[15] J. Hayes and C. Gutierrez. Bipartite graphs as intermediate model for RDF. In *Third International Semantic Web Conference (ISWC2004)*, vol. 3298 of *Lecture Notes in Computer Science*, pp. 47 – 61. Springer-Verlag, Hiroshima, Japan, November 2004.

[16] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, 2004.

[17] I. Herman. W3C Semantic Web Activity. `http://www.w3.org/2001/sw/`.

[18] M. A. W. Houtsma and A. N. Swami. Set-oriented mining for association rules in relational databases. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pp. 25–33. IEEE Computer Society, Washington, DC, USA, 1995.

[19] J. Kahan, M. Koivunen, E. Prud'Hommeaux, and R. Swick. Annotea: An open RDF infrastructure for shared web annotations. In *WWW Conf.*, pp. 623–632. 2001.

[20] G. Klyne and J. J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. Recommendation, W3C, February 2004. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.

[21] B. Lund, T. Hammond, M. Flack, and T. Hannay. Social Bookmarking Tools (II): A Case Study - Connotea. *D-Lib Magazine*, 11(4), April 2005.

[22] A. Maedche and S. Staab. Semi-automatic engineering of ontologies from text. In *Proceedings of the 12th Internal Conference on Software and Knowledge Engineering. Chicago, USA.* July 2000.

[23] F. Manola and E. Miller. RDF primer. Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[24] C. Marlow, M. Naaman, D. Boyd, and M. Davis. HT06, tagging paper, taxonomy, flickr, academic article, to read. In *HYPERTEXT '06: Proceedings of the seventeenth conference on Hypertext and hypermedia*, pp. 31–40. ACM Press, New York, NY, USA, 2006.

[25] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[26] N. F. Noy and M. A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 450–455. AAAI Press / The MIT Press, 2000.

[27] E. Oren. SemperWiki: a semantic personal Wiki. In *Semantic Desktop (ISWC)*. Nov. 2005.

[28] E. Oren, J. G. Breslin, and S. Decker. How semantics make better wikis. In *WWW (poster)*. May 2006.

[29] E. Oren, R. Delbru, S. Gerke, A. Haller, *et al.* ActiveRDF: Object-oriented semantic web programming. In *Proceedings of the International World-Wide Web Conference*. May 2007.

[30] E. Oren, M. Völkel, J. G. Breslin, and S. Decker. Semantic wikis for personal knowledge management. In *DEXA*. Sep. 2006.

[31] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[32] P. Resnick and H. R. Varian. Recommender systems. *Comm. of the ACM*, 40(3):56–58, 1997.

[33] L. Sauermann and S. Schwarz. Gnowsis adapter framework: Treating structured data sources as virtual RDF graphs. In *International Semantic Web Conference*, pp. 1016–1028. 2005.

[34] S. Schaffert. Ikewiki: A semantic wiki for collaborative knowledge management. *wetice*, 0:388–396, 2006.

[35] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[36] R. Srikant and R. Agrawal. Mining generalized association rules. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pp. 407–419. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[37] A. B. Tassilo Pellegrino. *Semantic Web. Wege zur vernetzten Wissensgesellschaft*. Springer, Berlin, 2006.

[38] M. Völkel, M. Krötzsch, D. Vrandevic, H. Haller, *et al.* Semantic wikipedia. In *WWW*. 2006.

[39] Wikipedia. Semantic Web. `http://en.wikipedia.org/wiki/Semantic_Web`, January 2007.

Erklärung

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 1. Juni 2007