OntoLiFT Demonstrator

WonderWeb: Ontology Infrastructure for the Semantic Web

Raphael Volz[1], Siegfried Handschuh[1], Steffen Staab[1], Rudi Studer[1,2]

[1]University of Karlsruhe
Institute AIFB
D-76128 Karlsruhe
email: {lastname}@aifb.uni-karlsruhe.de

[2]FZI - Research Center for Information Technologies
Haid-und-Neu-Strasse 10-14
D-76131 Karlsruhe
email: {lastname}@fzi.de

Identifier	Del 12
Class	Deliverable
Version	1.0
Date	29-04-2004
Status	Final
Distribution	Public
Lead Partner	AIFB

WonderWeb Project

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project co-ordinator:

Ian Horrocks
The Victoria University of Manchester
Department of Computer Science
Kilburn Building
Oxford Road
Manchester M13 9PL
Tel: +44 161 275 6154
Fax: +44 161 275 6236
Email: wonderweb-info@lists.man.ac.uk

Contents

Executive Summary 1

1 Introduction 2

2 Use Cases for Deep Annotation 4

3 The Process of Deep Annotation 5

4 Architecture 6

5 Server-Side Web Page Markup 7
5.1 Requirements 8
5.2 Database Representation 8
5.2.1 Formal model 8
5.2.2 Modelling considerations 9
5.2.3 An example representation 10
5.3 Query Representation 11
5.4 Result Representation 12

6 Create Mappings by Annotation of the Web Presentation 12
6.1 Annotation Process 12
6.2 Creating Generic Instances of Concepts 13
6.3 Creating Generic Attribute Instances 14
6.4 Creating Generic Relationship Instances 15

7 Create Mappings by Annotation of the Schema 15
7.1 Annotation Process 15
7.2 Schema to Ontology Mapping 15
7.2.1 Creating Concept Mappings 16
7.2.2 Creating Relationship Mappings 18
7.3 Demonstrator implementation 20

8 Accessing the Database 20
8.1 Querying the Database 20
8.2 Data Migration 21
8.3 Data Transformation 21

9 Related Work 22
9.1 Information Integration 22
9.2 Mapping and Merging Frameworks 22
9.3 Wrapper Construction 23
9.4 Database Reverse Engineering 23
9.5 Annotation 23

10 Conclusion 24

Executive Summary

This deliverable describes the demonstrator of OntoLift. We advanced the prototype of OntoLift, which was described in deliverable D11, with respect to lifting instance data from dynamically-generated Web pages. We thereby complement the previously described technique to derive ontologies from relational database schemata with a formation of the knowledge base of the ontology.

Previous versions of this report were accepted for publication in the prestigious WWW 2003 conference[1] (cf. [12]) and the Web Semantics journal (cf. [30]), which is the leading scientific journal in the Semantic Web area.

[1] Acceptance rate 13%.

1 Introduction

One of the core challenges of the Semantic Web is the creation of metadata by mass collaboration, i.e. by combining semantic content created by a large number of people. To attain this objective several approaches have been conceived (e.g. CREAM [9], MnM [29], or Mindswap [8]) that deal with the manual and/or the semi-automatic creation of metadata from existing information. These approaches, however, as well as older ones that provide metadata, e.g. for search on digital libraries, build on the assumption that the information sources under consideration are *static*, e.g. given as static HTML pages or given as books in a library (cf., [A,B] in Table 1). Such static information must not necessarily be embedded in the HTML page (case α in Table 1) but might also be stored remotely on some other server (case β in Table 1).

Nowadays, however, a large percentage of Web pages are not static documents. On the contrary, the majority of Web pages are dynamic.[2] For dynamic web pages (e.g. ones that are generated from the database that contains a bibliography) it does not seem to be useful to manually annotate every single page. Rather one wants to "annotate the database" in order to reuse it for one's own Semantic Web purposes.

For this objective, approaches have been conceived that allow for the construction of wrappers by explicit definition of HTML or XML queries [23] or by learning such definitions from examples [13, 3]. Thus, it has been possible to manually create metadata for a set of structurally similar Web pages. The wrapper approaches come with the advantage that they do not require cooperation by the owner of the database. However, their shortcoming is that the correct scraping of metadata is dependent to a large extent on data layout rather than on the structures underlying the data (cf., [C] in Table 1).

While for many web sites and underlying databases, the assumption of non-cooperativity may remain valid, we assume that many web sites will in fact participate in the Semantic Web and will support the sharing of information. Such web sites may present their information as HTML pages for viewing by the user, but they may also be willing to give (partial) access to the underlying database and to describe the structure of their information on the very same web pages. Thus, they give their users the possibility to utilize

1. information proper,
2. information structures (e.g., the logical database schema of a relational database), and
3. information context (e.g., the presentation into which information retrieved from the database is included).

A user may then exploit these three in order to create mappings into his own information structures (e.g., his ontology) and/or to migrate the data into his own repository. — which

[2] It is not possible to give a percentage of dynamic to static web pages in general, because a single Web site may use a simple algorithm to produce an infinite number of, probably not very interesting, web pages. Estimations, however, based on web pages actually crawled by existing search engines estimate that dynamic web pages outnumber static ones by 100 to 1.

may be a lot easier than if the information a user receives is restricted to information structures [19] and/or information proper alone [6].

We define "*deep annotation*" as an annotation process that reaches out to the so-called *Deep Web*[3] in order to make data available for the Semantic Web— combining the capabilities of conventional annotation and databases.

Table 1 summarizes the different settings just laid out.

Table 1: Principal situation

web site	cooperative owner	uncooperative owner
static	(α) Embedded or (β) remote metadata by conventional annotation [A]	Remote metadata by conventional annotation [B]
dynamic	*Deep annotation* with (α) server-side mapping rules or (β) client-side mapping rules or migrated data [D]	Wrapper construction, remote metadata [C]

Thereby, Table 1 further distinguishes between two scenarios regarding static web sites. In the one scenario [B], the annotator is not allowed to change static information, but he can create the metadata and remotely retain it from the source it belongs to (e.g., by XPointer). In the other scenario [A], he is free to choose between embedding the metadata created in the annotation process into the information proper (α; e.g., via the <meta> tag of a HTML page) or keeping it remote (β).[4] For deep annotation [D] the two choices boil down to either storing a created mapping within the database of the server (α) or to storing mapping and/or migrated data remotely from the server (β).

In the remainder of the deliverable, we will describe the building blocks for deep annotation. First, we elaborate on the use cases of deep annotation in order to illustrate its possible scope (Section 2). One of the use cases, a portal, will serve as a running example in the remainder of the deliverable. We continue with a description of two possible scenarios for using the deep annotation process in Section 3). These two scenarios exploit a tool set described in the architecture Section 4). The tool set exploits

1. the description of the database given as a complete logical schema or as a description of the database by server-side web page markup that defines the relationship between the database and the web page content (cf. Section 5);

2. annotation tools to actually let the user utilize the description of queries underlying a dynamic Web page (cf. Section 6) or the database schema underlying a dynamic Web site (cf. Section 7) for mapping information.

3. Components that let the user exploit the mappings, allowing him to investigate the constructed mappings (cf. Section 8), and query the serving database or to migrate data into his own repository.

[3] See HTTP://LIBRARY.ALBANY.EDU/INTERNET/DEEPWEB.HTML for a discussion of the term 'deep web'.

[4] Cf. [9] on those two possibilities.

Before we conclude with future work, we summarize our lessons learned and relate our work to other communities that have contributed to the overall goal of metadata creation and exploitation.

2 Use Cases for Deep Annotation

Deep annotation is relevant for a large and fast growing number of dynamic web sites that aim at cooperation, for instance:

Scientific databases. They are frequently built to foster cooperation among researchers. Medline, Swissprot, or EMBL are just a few examples that can be found on the Web. In the bio informatics community alone current estimations are that 500+ large databases are freely accessible.

Such databases are frequently hard to understand and it is often difficult to evaluate whether the table named “gene” in one database corresponds to the table name “gene” in the other database. For example, [24] reports an interesting case of semantic mismatches, since even an - apparently - unambiguous term like gene may be conceptualized in different ways in different genome databases. According to one (GDB), a gene is a DNA fragment that can be transcribed and translated into a protein, whereas for others (Genbank and GSDB), it is a “DNA region of biological interest with a name and that carries a genetic trait or phenotype”. Hence, it may be much easier to tell the meaning of a term from the context in which it is presented than from the concrete database entry, for example if genes are associated with the proteins they encode.

Syndication. Besides direct access to HTML pages of news stories or market research reports, etc., commercial information providers frequently offer syndication services. The integration of such syndication services into the portal of a customer is typically expensive manual programming effort that could be reduced by a deep annotation process that defines the content mappings.

For the remainder of the deliverable we will focus on the following use case:

Community Web Portal (cf., [25]). This serves the information needs of a community on the Web with possibilities for contributing and accessing information by community members. A recent example that is also based on Semantic Web technology is[5] [27]. The interesting aspect to such portals lies in the sharing of information, and some of them are even designed to deliver semantic information back to their community as well as to the outside world.[6]

The primary objective of a community setting up a portal will continue to be the opportunity of access for human viewers. However, given the appropriate tools they could easily provide information content, information structures and information context to their members for deep annotation. The way that this process runs is described in the following.

[5] http://www.ontoweb.org

[6] Cf., e.g., [26] for an example producing RDF from database content.

3 The Process of Deep Annotation

The process of creating deep annotation consists of the following four steps (depicted in Figure 1):

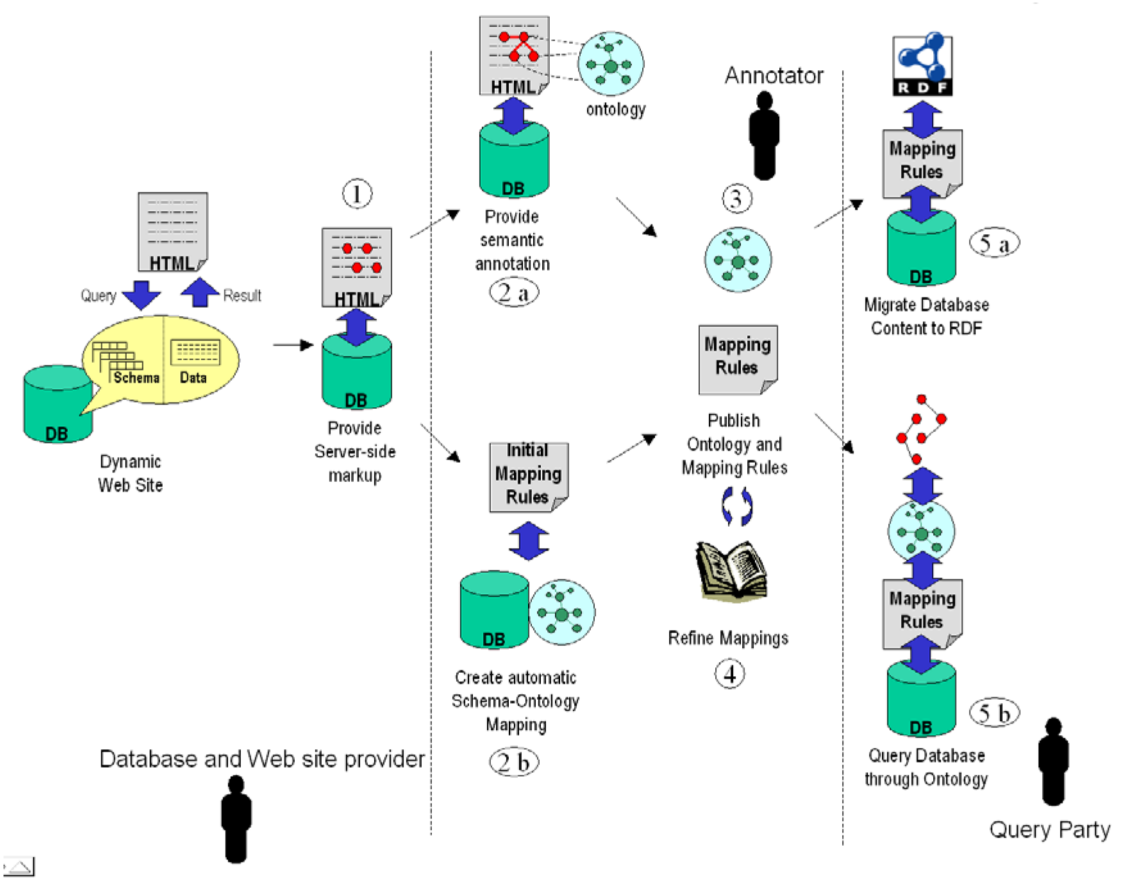

HTML
Query
Result
Schema
Data
DB
Dynamic Web Site
Database and Web site provider
1
HTML
DB
Provide Server-side markup
HTML
ontology
DB
Provide semantic annotation
2 a
Initial Mapping Rules
DB
Create automatic Schema-Ontology Mapping
2 b
Annotator
3
Mapping Rules
Publish Ontology and Mapping Rules
Refine Mappings
4
R D F
Mapping Rules
DB
5 a
Migrate Database Content to RDF
Mapping Rules
DB
5 b
Query Database through Ontology
Query Party

Figure 1: The Process of Deep Annotation

Input: A Web site[7] driven by an underlying relational database.

Step 1: The database owner produces server-side web page markup according to the information structures of the database (described in detail in Section 5).

Result: Web site with server-side markup.

Step 2: The annotator produces client-side annotations conforming to the client ontology either via web presentation-based annotations (Step 2a, cf. Section 6) or via automatic schema to ontology mapping (Step 2b, cf. Section 7).

Result: Mapping rules between database and client ontology

[7] Cf. Section 10 on other information sources.

Step 3: The annotator publishes the client ontology (if not already done before) and the mapping rules derived from annotations (Section 7).

Result: The annotator's ontology and mapping rules are available on the Web

Step 4: The annotator assesses and refines the mapping using certain guidelines (Section 7).

Result: The annotator's ontology and refined mapping rules are available on the Web

Deep annotations can be used in two ways (depicted in Figure 1): Firstly, the querying party loads second party's ontology and mapping rules and uses them to query the database via the database interface (e.g. a Web Service API) (cf. Section 8.1). Secondly, the querying party maybe interested in a migration of the complete (mapped) database content to ontology-based RDF instance data (e.g. in a RDF store such as Sesame [2]).

4 Architecture

Our architecture for deep annotation consists of three major pillars corresponding to the three different roles (database owner, annotator, querying party) as described in the process.

Database and Web Site Provider. At the web site, we assume that there is an underlying database (cf. Figure 2) and a server-side scripting environment, e.g. Zope or Java Server Pages (JSP), used to create dynamic Web pages. The database also has to provide some Web accessible interface (e.g. a web service API) allowing third parties to query the database directly.

Furthermore, the information about the underlying database structure, which is necessary for the mapping of the database, will be available on the web pages and via the API.

Annotator. The annotator has two choices for the deep annotation of the database: i) indirectly, by annotation of the web presentation or ii) directly by annotation of the logical database schema.

For the annotation of the web presentation the annotator uses OntoLift, which is an extended version of the OntoMat-Annotizer, in order to manually create relational metadata, which correspond to a given client ontology, for some Web pages). OntoLift takes into account problems that may arise from generic annotations required by deep annotation (see Section 6). With the help of OntoLift, we create mapping rules from such annotations (route 1-2a-3-4 in the Figure 1 that are later exploited by an inference engine.

Alternatively, the annotator can base his mappings directly on the database scheme. In this case the input of the migration is an extended relational model that is derived from the SQL DDL. The database schema is mapped into the given ontology using the mapping process described below, which applies the rules specified in 7. The same holds for database instances that are transformed into a knowledge base, which is based on the domain ontology.

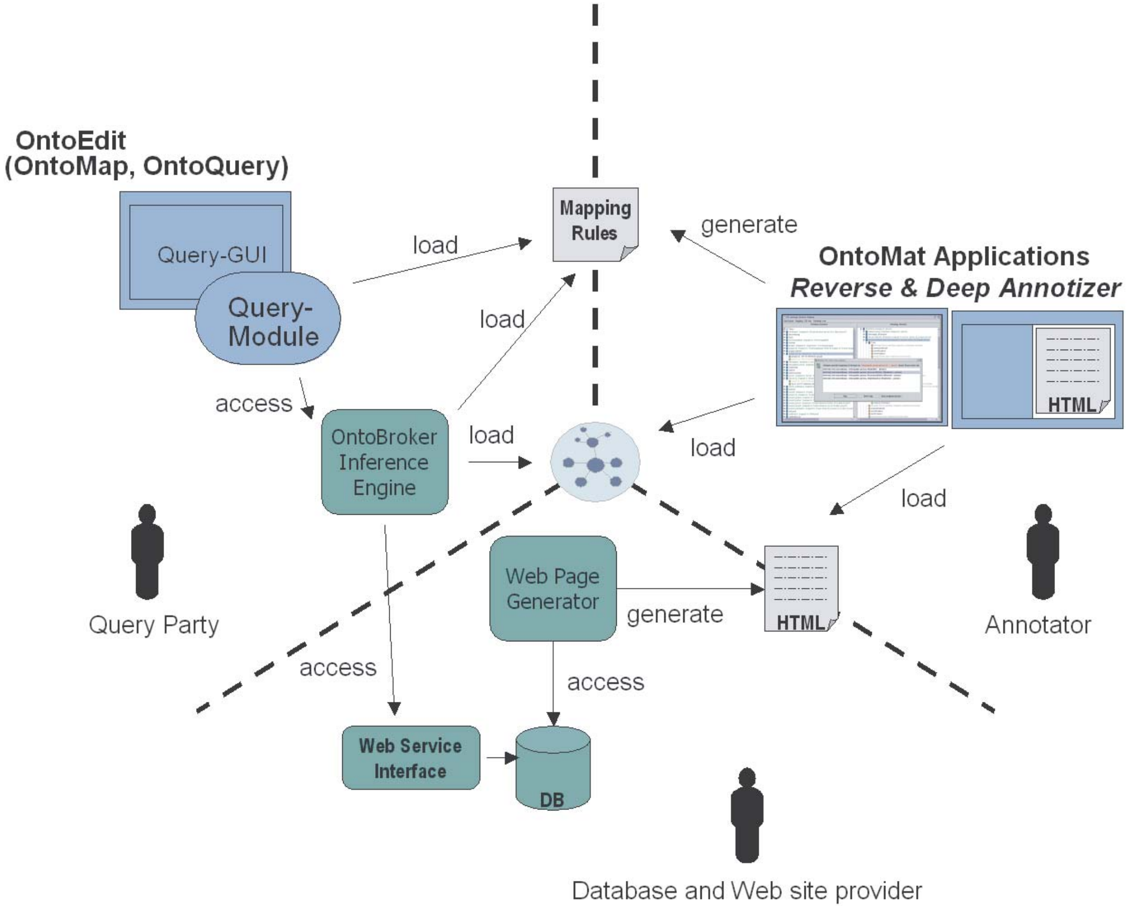


Figure 2: An Architecture for Deep Annotation

For the automation of the mapping process based on the schema of the underlying database (route 1-2b-3-4 in the Figure 1), we used OntoMat-Reverse, a tool for semi-automatically connecting relational databases to ontologies, which enables less experienced users to perform this mapping process. Moreover, OntoMat-Reverse automatizes some phases in that mapping process, particularly capturing information from the relational schema, validation of the mapping process and data migration (see Section 7.3).

Querying Party. The querying party uses a corresponding tool to visualize the client ontology, to compile a query from the client ontology and to investigate the mapping. In our case, we use OntoEdit [28] for those three purposes. In particular, OntoEdit also allows for the investigation, debugging and change of given mapping rules. To that extent, OntoEdit integrates and exploits the Ontobroker [7] inference engine (see Figure 2).

5 Server-Side Web Page Markup

The goal of the mapping process is to allow interested parties to gain access to the source data. To this extend pointers to the underlying data sources are required. The role of the

Server-Side Web Page Markup is exactly that, i.e. it describes the structure of the database schema and queries issued to the database from a certain page. This way one can obtain all information required to access the data from outside.

5.1 Requirements

All required information has to be published, so that an interested party can use this information to retrieve the data from the underlying database. The information which must be provided is as follows: *(i)* which database is used as a data source, how is this database structured, and how can it be accessed; *(ii)* which query is used to retrieve data from the database; and *(iii)* which elements of the query result are used to create the dynamic web page. Those three components are detailed in the remainder of this section.

5.2 Database Representation

The database representation is specified using a dedicated deep annotation ontology, which is instantiated to describe the physical structure of the part of the database which may facilitate the understanding of the query results. Thereby, the structure of all tables/views involved in a query can be published.[8]

5.2.1 Formal model

The deep annotation ontology contains concepts and relations such that the formal model of a database schema, viz. the relational model, can be expressed using RDF. In our ontological description we extend the common formal definition of the relational model with additional constructs typically found in SQL-DDLs, i.e. constructs which allow to state inclusion dependencies [**?**]. Hence, our ontology captures the following formal model:

Definition 1 *A relational schema S is a 8-tuple* $(R, A, T, I, att, key, type, notnull)$ *with:*

1. *A finite set R called Relations,*
2. *A finite set A called Attributes,*
3. *A function* $att : R \rightarrow 2^A$ *which defines the attributes contained in a specific relation* $r_i \in R$*,*
4. *function* $key : R \rightarrow 2^A$ *that defines which attributes are primary keys in a given relation (thus* $key(r_i) \subseteq att(r_i)$ *must hold),*
5. *A set* T *of atomic data types,*
6. *A function* $type : A \rightarrow T$ *that states the type of a given attribute,*

[8] The reader may note, that alternatively also the structure of the complete database can be published once and the description for an individual page can refer to this description via OWL imports mechanisms.

7. *A function notnull* $: R \rightarrow 2^A$ *which states those attributes of a relation which have to have a value,*

8. *A set of inclusion dependencies I where*

    - *each element has the form* $((r_1,A_1),(r_2,A_2))$,
    - $r_1, r_2 \in R$,
    - $A_1 = \{a_{11}, a_{12}, ..., a_{1n}\}$,
    - $A_2 = \{a_{21}, a_{22}, ..., a_{2n}\}$,
    - $A_1 \subseteq att(r_1)$ *and* $A_2 \subseteq att(r_2)$,
    - $\mid A_1 \mid = \mid A_2 \mid$ *and* $type(a_{1i}) = type(a_{2i})$

Remark 2

1. *We will refer to* r_1 *as domain relation and* r_2 *as range relation.*

2. I_c *denotes the transitive closure of* I.

5.2.2 Modelling considerations

The reader may note that SQL-DDLs are typically more expressive than relational algebra. For instance, it is usually possible to specify constraints (such as DEFAULT and NOT NULL). Default values for datatypes are not supported in current Web ontology languages and therefore ignored. The NOT NULL constraint is translated to the function ”notnull” and expressed in the ontology as a concept ”NotNullAttribute” that is subsumed by Attribute. Please note that SQL enforces automatically that $\forall r_i \in R : key(r_i) \subseteq notnull(r_i) \subseteq att(r_i)$.

In our modelling we do not consider the dynamic aspects found in SQL-DDLs for the deep annotation. Thus, triggers, referential actions (like ON UPDATE etc.) and assertions are not mapped.

In SQL-DDLs it is also possible to specify referential integrity constraints, by means of so-called foreign keys. This information is especially useful for the mapping process as it indicates associations between database relations. SQL referential integrity constraints reinforce the view that inclusion dependencies [**?**] are valid at all times.

With respect to inclusion dependencies, a pitfall in the translation process becomes apparent: The database designer is supposed to express associations between database relations by means of foreign keys to make these semantics explicit. However, the processing of this semantics is usually not supported by its definition. The combination of information supported via such associations is created manually by stating appropriate joins of tables in the queries to the database. Hence, those associations remain often unspecified. The appropriate semantics may only be extracted by analyzing the queries sent to the database. However, the latter is not feasible since running information systems would have to be altered to track the queries issued by the system. We therefore allow users to specify foreign keys a-posteriori.

5.2.3 An example representation

For example the following representation (that corresponds to the database schema shown in Figure 3) is a part of the HTML head of the web page presented in Figure 4.

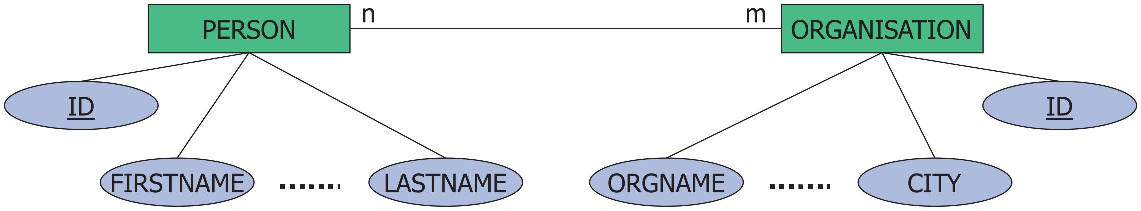


Figure 3: Example database schema

```
<!--

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:da="http://annotation.semanticweb.org#deepanno">
  <da:DB rdf:ID="OntoSQL">
    <da:accessService
        rdf:resource="www.ontoweb.org/database_access.wsdl"/>
  </da:DB>
  <da:Table rdf:ID="Person">
    <da:name>Person</da:sqlName>
    <da:inDatabase rdf:resource="#OntoSQL" />
    <da:hasColumns rdf:parseType="Collection">
        <da:PrimaryKey rdf:ID="Person.ID"
            da:name="ID" da:type="int" />
        <da:Column da:name="FIRSTNAME" da:type="varchar"/>
        <da:Column da:name="LASTNAME" da:type="varchar"/>
    </da:hasColumns>
  </da:Table>
  <da:Table rdf:ID="Organization">
    <da:name>Organization</da:name>
    <da:inDatabase rdf:resource="#OntoSQL" />
    <da:hasColumns rdf:parseType="Collection" />
        <da:PrimaryKey rdf:ID="Organization.ID"
            da:name="ID" da:type="int" />
        <da:Column da:name="ORGNAME" da:type="varchar"/>
        <da:Column da:name="CITY" da:type="varchar"/>
        ...
    </da:hasColumns>
  </da:Table>
  <da:Table rdf:ID="PersonOrg">
    <da:name>Person_Org<da:name>
    <da:inDatabase rdf:resource="#OntoSQL" />
    <da:hasColumns rdf:parseType="Collection" />
        <da:PrimaryKey da:name="PERSONID" da:type="int">
            <references rdf:resource="#Person.ID"/>
        </da:PrimaryKey>
        <da:PrimaryKey da:name="ORGID" da:type="int">
            <references rdf:resource="#Organization.ID"/>
        </da:PrimaryKey>
    </da:hasColumns>
  </da:Table>
</rdf:RDF>
-->
```

The RDF property da:accessService provides a link to the service which allows for anonymous database access. Consequently additional security measures can be implemented in this access service. For example, anonymous users should usually have read-access to public information only.

5.3 Query Representation

Additionally, the query itself, which is used to retrieve the data from a particular source is placed in the header of the page. It contains the intended SQL-query and is associated with a name as a means to distinguish between queries and operates on a particular data source.

```
<!--

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:da="http://annotation.semanticweb.org#deepanno">
    <da:Query rdf:ID="Q1">
      <da:source rdf:resource="#OntoSQL" />
      <da:hasResultColumns rdf:parseType="Collection">
        <ColumnGroup rdf:ID="g1" />
        <ColumnGroup rdf:ID="g2" />
      </da:hasResultColumns>
      <da:sql>
      SELECT Person.*, Organization.*
      FROM   Person, Organization, Projekt_Org
      WHERE  Person.ID = Projekt_Org.PERSONID
             AND Organization.ID = Projekt_Org.ORGID
      </da:sql>
    </da:Query>
    <da:Columngroup rdf:ID="g1">
        <da:prefix
            rdf:resource="http://www.ontoweb.org/person/">
        <da:hasColumns rdf:parseType="Collection">
            <Identifier da:name="Id" />
            <Column da:name="Firstname" />
            <Column da:name="Lastname" />
        </da:hasColumns>
    </da:Columngroup>
    <da:Columngroup rdf:ID="g2">
        <da:prefix
            rdf:resource="http://www.ontoweb.org/org/">
        <da:hasColumns rdf:parseType="Collection">
            <Identifier da:name="OrganizationId" />
            <Column da:name="Orgname" />
            <Column da:name="City" />
        </da:hasColumns>
    </da:Columngroup>
  </rdf:RDF>
-->
```

The structure of the query result must be published by means of column groups. Each column group must have at least one identifier, which is used in the annotation process to distinguish individual instances and detect their equivalence. Since database keys are only local to the respective table, but the Semantic Web has a global space of identifiers, appropriate prefixes have to established. The prefix also ensures that the equality of instance data generated from multiple queries can be detected, if the web application maintainer chooses the same prefix for each occurrence of that *id* in a query. Eventually, database keys are translated to instance identifiers (cf. Section 8.1) via the following pattern:

$$< prefix > [key_i - name = key_i - value]$$

For example: http://www.ontoweb.org/person/id=1

5.4 Result Representation

Whenever parts of the query results are used in the dynamically generated web page, the generated content is surrounded by a tag, which carries information about which column of the result tuple delivered by a query represents the used value. In order to stay compatible with HTML, we used the <span> tag as an information carrier. The actual information is represented in attributes of <span>:

```
<table> <tr> <td> <span da:qresult="q1"
da:column="Orgname">AIFB</span> </td>

<td> <span da:qresult="q1" da:column="City">Karlsruhe</span> </td>

... <td> <span da:qresult="q1"
da:column="Firstname">Steffen</span> </td>

...

</tr> </table>
```

Such span tags are then interpreted by the annotation tool and are used in the mapping step.

6 Create Mappings by Annotation of the Web Presentation

The annotator has two choices for the deep annotation of the database: i) indirectly, by annotation of the web presentation or ii) directly by annotation of the logical database schema. In this section we present the indirect way (route 1-2a-3-4 in Figure 1), namely to create the mappings by annotating individual pages, viz. the queries issued to the database when a certain page is dynamically constructed. The results of the annotation presented here are mapping rules between the database and the client ontology.

6.1 Annotation Process

An annotation in our context is a set of instantiations related to an ontology and referring parts of an (HTML) document. We distinguish *(i)* instantiations of concepts, *(ii)* instantiated properties from one concept instance to a datatype instance — henceforth called attribute instance (of the concept instance), and *(iii)* instantiated properties from one concept instance to another concept instance — henceforth called relationship instance.

In addition, for the deep annotation one must distinguish between a *generic annotation* and a *literal annotation*. In a *literal annotation*, the piece of text may stand for itself.

In a *generic annotation*, a piece of text that corresponds to a database field and that is annotated is only considered to be a place holder, i.e. a variable must be generated for such an annotation and the variable may have multiple relationships allowing for the description of general mapping rules. For example, a concept Institute in the client ontology may correspond to one generic annotation for the Organization identifier in the database.

Consequential to the above terminology, we will refer to generic annotation in detail as *generic concept instances*, *generic attribute instances*, and *generic relationship instances*.

An annotation process of server-side markup (generic annotation) is supported by the user interface as follows:

1. In the browser the user opens a server-side marked up web page.

2. The server-side markup is handled individually by the browser, e.g. it provides graphical icons on the page wherever a markup is present, so that the user can easily identify values which come from a database.

3. The user can select one of the server-side markups to either create a new *generic instance* and map its database field to a generic attribute, or map a database field to a *generic attribute* of an existing *generic instance*.

4. The database information necessary to query the database in a later step is stored along with the *generic instance*.

The reader may note that *literal annotation* is still performed when the user drags a marked-up piece of content that is not a server-side markup.

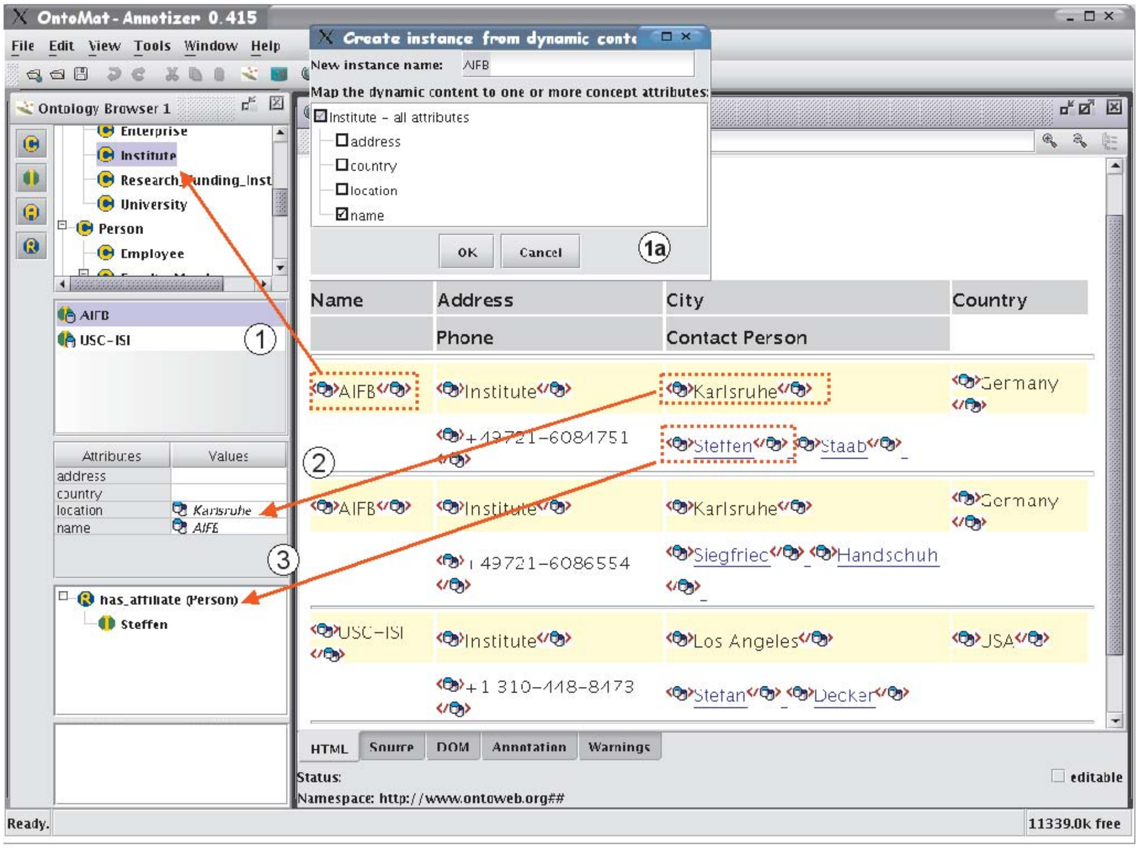
6.2 Creating Generic Instances of Concepts

When the user drags an item identified as server-side markup onto a particular concept of the ontology, a new generic concept instance is generated (cf. arrow #1 in Figure 4). A dialog is then presented to the user to capture the instance name and its attributes to which the database values should to be mapped. Attributes which resemble the column name are preselected (cf. dialog #1a in Figure 4). If the user confirms this operation, database concept and instance checks are performed and the new generic instance is created. Generic instances will later appear with a database symbol in their icon.

Along with each generic instance the information about its underlying database query and the unique identifier pattern is stored. This information is retrieved from the markup at creation time of the instance as each server-side markup contains a reference to the query, the selected column, and the value of the column. The identifier pattern is obtained from the reference to the query description and the according column group (cf. Section 5.3). The markup used to create the instance, defines the identifier pattern for the generic instance. The identifier pattern will be used when instances are generated from the database (cf. Section 8.1).

Let’s consider the example displayed in Figure 4: At the beginning the user selects the server-side markup ”AIFB” and drops it on the concept Institute. The content of the server-side markup is ’<span qresult=”q1” column=”Orgname”>AIFB</span>’. This creates a new generic instance with a reference to the query *q1* (cf. Section 5.3). Immediately after this action the users chooses to use the values of database column ”OrgName” as a filler of the generic concept attribute ”name” for all generic instances ”AIFB”. The identifier pattern for generic instances is created as a side effect of this action. In our example this is

HTTP://WWW.ONTOWEB.ORG/ORG/ORGANIZATIONID=ORGANIZATIONID,
since ”OrganizationID” is the database column which presents the primary key in query *q1*.



Create instance from dynamic conte

New instance name: AIFB

Map the dynamic content to one or more concept attributes:

☒ Institute – all attributes
- □ address
- □ country
- □ location
- ☑ name

OK Cancel

Attributes	Values
address	
country	
location	*Karlsruhe*
name	*AIFB*

Name	Address	City	Country
	Phone	Contact Person	
AIFB	Institute	Karlsruhe	Germany
	+49721-6084751	Steffen Staab	
AIFB	Institute	Karlsruhe	Germany
	+49721-6086554	Siegfriec Handschuh	
USC-ISI	Institute	Los Angeles	USA
	+1 310-448-8473	Stefan Decker	

Figure 4: Screenshot of Providing Deep Annotation with OntoMat-DeepAnnotizer

6.3 Creating Generic Attribute Instances

The user simply drags the server-side markup into the corresponding table entry (cf. arrow #2 in Figure 4) to create a generic attribute instances. If the dragged content does not contain server-side markup, the content itself will be used as a default value (that is shared by all generic instances).

If the content contains server-side markup the following steps are performed. Firstly, all generic attributes which are mapped to database table columns will also show a special icon. The current value of the dragged content will be used for display purposes in the interface where it appears in italic font. In this case the assignment of generic attributes can be removed but the exemplary value itself is immutable.

The following steps are additionally performed by the system when the generic attribute is filled:

1. The integrity of the database definition in the server-side markup is checked.

2. We check whether the generic attribute is selected from the same query as used for the generic instance. This ensures that result fields come from the same database and the same query otherwise non-matching information (e.g. publication titles and countries) could be queried.

3. All information given by the markup, i.e. which column of the result tuple delivered by the query represents the value, is associated with the generic attribute instance.

6.4 Creating Generic Relationship Instances

In order to create a generic relationship instance the user simply drops the selected server-side markup onto the relation of a pre-selected instance (cf. arrow #3 in Figure 4). As in Section 6.2 a new generic instance is generated. In addition, the new generic instance is connected with the pre-selected generic instance.

7 Create Mappings by Annotation of the Schema

In this section we consider the second alternative (route 1-2b-3-4 in Figure 1), namely to annotate the database schema and base the annotations directly on the schema. While this has the benefit, that one can annotate a whole site that origins from a database, it comes with the drawback that we miss the information context, e.g. the presentation into which information in a web page is included.

7.1 Annotation Process

An annotation process of server-side markup is supported by the user interface as follows:

1. In the browser the user opens a server-side marked up web page.

2. The database schema description included in the server-side markup is loaded by the tool.

3. The user can ask the tool to automatically create an initial mapping to the ontology based on the lexicalizations and the structure of the database (cf. the remainder of this section). This mapping is created in two phases, e.g. first concept mappings are created and afterwards relationship mappings are created.

4. The user may refine, remove and create mappings between relational schema and ontology as required.

The reader may note that no *literal annotation* is performed anymore, only *generic annotations* remain.

7.2 Schema to Ontology Mapping

The automatic schema to ontology mapping is carried out via mapping rules, that are applicable if formal preconditions are met. Mapping rules map elements of the server schema to entities in the client ontology. In the following presentation we use the auxiliary functions:

- $concept : R \rightarrow C$ to denote the association between a relation and a concept.
- $typetrans : T \rightarrow D$ transforming relational data types into the appropriate XML schema datatypes.

In order to discover mappings several approaches [22, 4, 6, 1, 19, 17], can be used. Our own approach is based on [16] and extended to incorporate the structural heterogeneity resulting from the fact that we do not map between ontologies but two different meta models, viz. relational schema and ontology. This extention is required to map the implicit semantics incorporated in some relational schemas to explicit ontological structures. The automatic translation generally depends on a lexical agreement of part of the two ontologies/database schemata. This dependency is weakened since users typically manually refine the obtained mapping at a later stage.

7.2.1 Creating Concept Mappings

Database relations are mapped to Concepts if a lexical agreement in the naming exists. However, since schemas usually incorporate many elements that are only defined to avoid the structural limitations of the relational model or made for performance reasons, certain relations are excluded from potential concept mappings.

n:m Associations Certain database relations are only defined to express (n:m) associations between two other relations. Typically, this type of auxiliary relation is characterized by the fact that it contains only two attributes, which are both primary keys and foreign keys to two other relations. A dedicated mapping rule excludes these relations from the mapping, hence as a result no concept mapping is created.

Translation Rule 1 ((n:m) Association) *Auxiliary relations used to specify (n:m) associations may be identified via the following conditions:*

- $att(r_i) = key(r_i)$
- $A_1 \subset att(r_i), A_2 \subset att(r_i)$
- $A_1 \cup A_2 = att(r_i)$
- $A_1 \cap A_2 = \emptyset$
- $((r_i, A_1), (r_j, key(r_j)) \in I$
- $((r_i, A_2), (r_k, key(r_k)) \in I$

A similar rule triggers the creation of a mapping to relationships that holds between both concepts c_j, c_k[9]. Preference is given to relationships that are mutually inverse to each other.

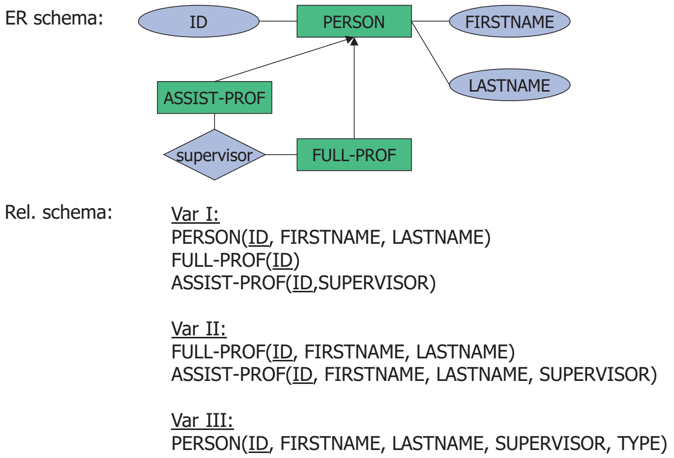
Information Distribution In certain cases information that logically corresponds to one entity in the ontology is distributed across several database relations for performance reasons. This is for example the case if one of the attributes is storage-intensive and optional. To optimize the clustering behavior of the database such attributes are often stored in separate relations together with the primary key of the main relation.

[9] where $c_x = concept(r_x)$

Translation Rule 2 (Information distribution) *Information distribution may be detected with the following heuristic:*

- $((r_i, key(r_i)), (r_j, key(r_j)) \in I$

As a result no concept mappings are created. Instead a similar rule triggers the aggregation of attributes of both relations and their conversion to relationship mappings on only one concept.



Var I:
PERSON(ID, FIRSTNAME, LASTNAME)
FULL-PROF(ID)
ASSIST-PROF(ID,SUPERVISOR)

Var II:
FULL-PROF(ID, FIRSTNAME, LASTNAME)
ASSIST-PROF(ID, FIRSTNAME, LASTNAME, SUPERVISOR)

Var III:
PERSON(ID, FIRSTNAME, LASTNAME, SUPERVISOR, TYPE)

Figure 5: Specialization in Extended Entity Relationship Diagrams

This heuristic is slightly problematic since it also covers one of the well-known mapping procedures for translating entity hierarchies in EER models to relational schemas. Hence, a similar heuristic could alternatively lead to two concept mappings given that a subsumption relation between those concepts holds.

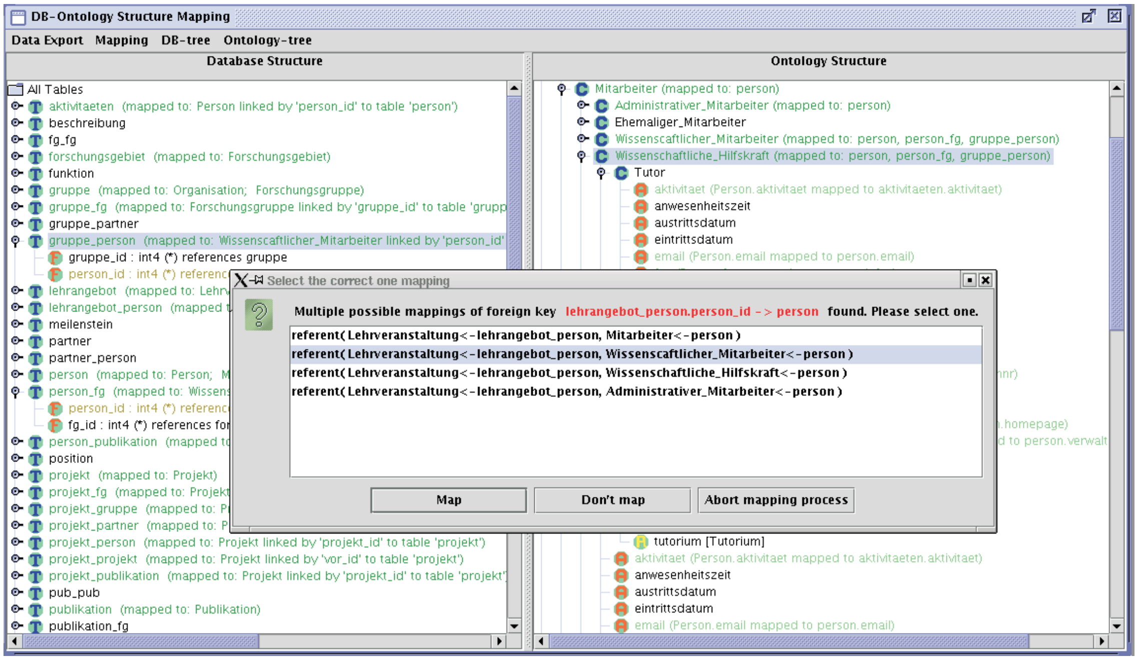
Figure 5 depicts such a situation. Here the entity PERSON has several specialized entities ASSIST-PROF and FULL-PROF. Usually this may be translated to relational schemas via the following principles [**?**]:

1. Create a relation for the super entity and relations for each sub entity, such that they contain all attributes of the sub entity and the primary key of the super entity.
2. If the specialization is total (there is no instance of the super entity, only sub entities are instantiated): Create a relation for all sub entities only and copy the attributes defined for the super entity.
3. If the specialization is disjoint (an instance may only be instance of one entity): Create only one relation that contains the attributes defined for all entities and makes those attributes optional. Add an extra attribute to explicitly state the type.

This clearly shows that the semantic intention behind a given relational structure cannot be captured fully by the automatism.

Translation Rule 3 *Our general heuristic is that each relation is mapped to the lexically closest concept.*

This heuristic is applied when no other rule could be applied. In the current tool edit distance [14] is used to identify the lexically closest concept. Other lexical distances, e.g. Hamming distance could be used as well but have not been tested within the tool. Along the same lines one could apply different preprocessing heuristics such as stemming. However, we did not experiment with such strategies yet.



```
All Tables
aktivitaeten  (mapped to: Person linked by 'person_id' to table 'person')
beschreibung
fg_fg
forschungsgebiet  (mapped to: Forschungsgebiet)
funktion
gruppe  (mapped to: Organisation;  Forschungsgruppe)
gruppe_fg  (mapped to: Forschungsgruppe linked by 'gruppe_id' to table 'grupp
gruppe_partner
gruppe_person  (mapped to: Wissenscaftlicher_Mitarbeiter linked by 'person_id'
    gruppe_id : int4 (*) references gruppe
    person_id : int4 (*) referenc
lehrangebot  (mapped to: Lehrv
lehrangebot_person  (mapped t
meilenstein
partner
partner_person
person  (mapped to: Person;  M
person_fg  (mapped to: Wissens
    person_id : int4 (*) referenc
    fg_id : int4 (*) references for
person_publikation  (mapped to
position
projekt  (mapped to: Projekt)
projekt_fg  (mapped to: Projekt
projekt_gruppe  (mapped to: Pr
projekt_partner  (mapped to: P
projekt_person  (mapped to: Projekt linked by 'projekt_id' to table 'projekt')
projekt_projekt  (mapped to: Projekt linked by 'vor_id' to table 'projekt')
projekt_publikation  (mapped to: Projekt linked by 'projekt_id' to table 'projekt'
pub_pub
publikation  (mapped to: Publikation)
publikation_fg
```

```
Mitarbeiter (mapped to: person)
  Administrativer_Mitarbeiter (mapped to: person)
  Ehemaliger_Mitarbeiter
  Wissenscaftlicher_Mitarbeiter (mapped to: person, person_fg, gruppe_person)
  Wissenschaftliche_Hilfskraft (mapped to: person, person_fg, gruppe_person)
    Tutor
      aktivitaet (Person.aktivitaet mapped to aktivitaeten.aktivitaet)
      anwesenheitszeit
      austrittsdatum
      eintrittsdatum
      email (Person.email mapped to person.email)
      nnr)
      .homepage)
      d to person.verwalt
      tutorium [Tutorium]
    aktivitaet (Person.aktivitaet mapped to aktivitaeten.aktivitaet)
    anwesenheitszeit
    austrittsdatum
    eintrittsdatum
    email (Person.email mapped to person.email)
```

Multiple possible mappings of foreign key lehrangebot_person.person_id -> person found. Please select one.

```
referent( Lehrveranstaltung<-lehrangebot_person, Mitarbeiter<-person )
referent( Lehrveranstaltung<-lehrangebot_person, Wissenscaftlicher_Mitarbeiter<-person )
referent( Lehrveranstaltung<-lehrangebot_person, Wissenschaftliche_Hilfskraft<-person )
referent( Lehrveranstaltung<-lehrangebot_person, Administrativer_Mitarbeiter<-person )
```

Figure 6: Automatic Schema to Ontology Mapping

7.2.2 Creating Relationship Mappings

Attributes vs. Relationships As mentioned before the OntoMat tool distinguishes between attributes and relationships[10]. The tool cannot decide whether a given attribute is only defined for organizational purposes, such as many auto-incremented primary keys, which do not carry any real meaning beyond tuple identification[11] and should therefore be excluded from the mapping. Consequently, all attributes are mapped to corresponding datatype properties, if such properties are defined for the concept or one of its subconcepts. The domain of attributes is naturally the concept, to which the relation upon which the attribute is defined is mapped.

Associations between database relations, which are expressed via foreign keys (and constitute inclusion dependencies), are mapped to object properties. Hence, inclusion dependencies are translated into mappings to object properties. The domain concept of an object property corresponds to the mapping target of the domain-relation in the inclusion dependency. The range concept corresponds to mapping target to the range-relation of the inclusion dependency, respectively.

[10] In OWL terminology: datatype properties and object properties.

[11] The role of the latter is provided by URIs in the Semantic Web, hence this information would no longer be needed.

Translation rules The default rule is to map an attribute to the lexically closest atrribute defined for a concept (or one of its subconcepts). The default rule will be applied if no other rules are applicable. Further mapping rules are used to create mappings to relationships between ontologies. In the following we use the following definitions across these mapping rules:

- $((r_i, A_1), (r_j, key(r_j)) \in I$
- $((r_i, A_2), (r_k, key(r_k)) \in I, r_j \neq r_k$
- $c_j = concept(r_j)$
- $c_k = concept(r_k)$

The mapping rules generally result in up to two mappings to the lexically closest relationships holding in either direction between c_j and c_k. Preference is given to relationships that are mutually inverse to each other, and have either c_j or c_k or one of their subconcepts as domain and range respectively. All other translation rules are applied in the following order:

Translation Rule 4 ((n:m) association) *The first heuristic takes care of (n:m) associations between database relations. The preconditions for the application of this heuristic are:*

- $A_1 \subset att(r_i), A_2 \subset att(r_i)$
- $att(r_i) = key(r_i)$
- $A_1 \cup A_2 = att(r_i)$
- $A_1 \cap A_2 = \emptyset$
- $((r_i, A_1), (r_j, key(r_j)) \in I$
- $((r_i, A_2), (r_k, key(r_k)) \in I, r_j \neq r_k$
- $c_j = concept(r_j)$
- $c_k = concept(r_k)$

Translation Rule 5 ((1:1) association) *This heuristic captures (1:1) associations between database relations. The preconditions for the applicability of this heuristic are:*

- $c_i = concept(r_i)$
- $c_j = concept(r_j)$
- $((r_i, key(r_i)), (r_j, key(r_j)) \in I$
- $((r_j, key(r_j)), (r_i, key(r_i)) \in I$

Translation Rule 6 ((1:m) association) *This heuristic treats one to many associations, which are constituted by foreign keys. The precondition for the application of this heuristic are:*

- $c_i = concept(r_i)$
- $c_j = concept(r_j)$

- $A_1 \subseteq att(r_i)$
- $((r_i, A_1), (r_j, key(r_j))) \in I$

Translation Rule 7 (Role Grouping) *This mapping rule is used to group the attributes that are distributed in several relations and complements rule 2. The preconditions for the applicability of this rule are:*

- $\neg \exists c_i = concept(r_i)$
- $\exists c_j = concept(r_j)$
- $((r_i, key(r_i)), (r_j, key(r_j))) \in I$

For all attributes $A = att(r_i) \setminus key(r_i)$ new roles with domain concept c_j are created.

7.3 Demonstrator implementation

The automatic mapping is prototypically implemented in the OntoMat-Reverse tool that also enables very intuitive presentation/inspection of the database's relational schema and the structure of the given ontology, as presented in Figure 6. OntoMat-Reverse uses edit distance [14] as a measure to obtain lexical matching hypotheses as required for the mapping rules. For example, the database relation named ”Projekt” can be mapped into the concept named ”Project”, since the edit distance between these words is very small.

Additionally OntoMat-Reverse supports a validation step: for example, it can discover that a database relation is not mapped into any concept. In that case, it analysis the structure and the content of such a database relation, in order to determine the cause of the problem, for example that this database relation has neither foreign keys nor that other database relations have a foreign key to this database relation.

Figure 6 shows how OntoMat-Reverse makes recommendations for assigning attributes of the database relation to ontological relations between concepts in the ontology. The left side of figure depicts the structure of the source database schema. The right side shows the target ontology. Highlighted entities are mapped to each other. Recommendations for mapping attributes are listed in the dialog.

8 Accessing the Database

Deep annotations can be used in two ways to access the database. Firstly, interested parties can query the database. Here, the querying party loads second party's ontology and mapping rules and uses them to obtain database tuples via the database interface. Secondly, interested parties can use the mapping to migration the complete (mapped) database into ontology-based RDF instance data.

8.1 Querying the Database

The querying party can use further tools to visualize the client ontology, to investigate the published mapping and to compile a query from the client ontology. In our case, we used the OntoEdit plugins OntoMap and OntoQuery.

OntoMap visualizes the database query, the structure of the client ontology, and the mapping between them (cf. Figure 7). The user can control and change the mapping and also create additional mappings.

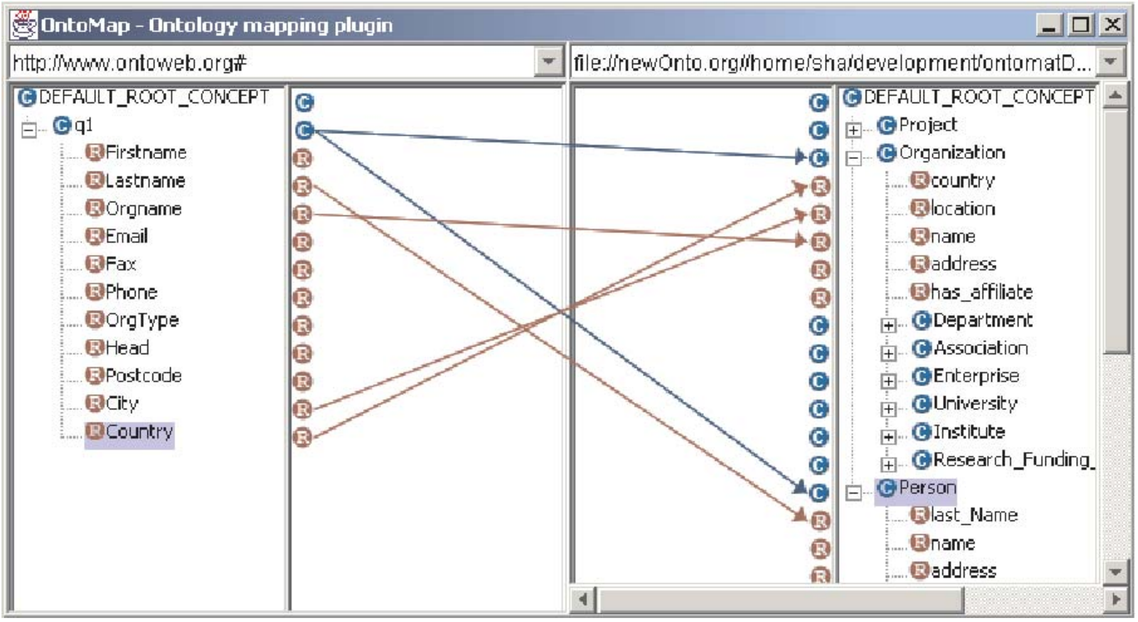


Figure 7: Mapping between Server Database (left window) and Client Ontology (right window)

OntoQuery is a Query-by-Example user interface. One creates a query by clicking on a concept and selecting the relevant attributes and relationships. The underlying Ontobroker system transforms a query on the ontology into a corresponding SQL query on the database. To this extend, Ontobroker uses the mapping descriptions, which are internally represented as F-Logic axioms, to transform the query. The SQL query is sent to the database via the interface published in the server-side markup. The database answer, viz. a set of tuples, is transformed back into the ontology-based representation using the mapping rules. This task is executed automatically, hence no interaction with the user is necessary.

8.2 Data Migration

Alternatively to querying the mappings can also be used to materialize ontology instances into RDF files. Here, all tuples of the relation database that are reachable via the mapping are stored explicitly and form a knowledge base for the published client ontology.

8.3 Data Transformation

Data has to be transformed to ontology-based data for both methods of accessing the database. This data transformation is executed in two separate steps. In the first step, all the required concept instances are created without considering relationships or attributes. These instances are stored together with their identifier. The identifier is translated from the database keys using the identifier pattern (see Section 5.2). For example, the instance with the name "AIFB" of the concept Institute, which is a subconcept of Organization, has the identifier: HTTP://WWW.ONTOWEB.ORG/ORG/ORGANIZATIONID=3.

After the creation of all instances the system starts computing the values of the instance relationships and attributes. The way the values are assigned is determined by the mapping rules. Since the values of an attribute or a relationship have to be computed from both the relational database and the ontology, we generate two queries per attribute/relationship, one SQL query and one Ontobroker query. Each query is invoked with an instance key value (corresponding database key in SQL-queries) as a parameter and returns the value of the attribute/relationship.

Note that the database communication takes place through binding variables. The corresponding SQL query is generated, and if this is the first call, it is cached. A second call would try to use the same database cursor if still available, without parsing the respective SQL statement. Otherwise, it would find an unused cursor and retrieve the results. In this way efficient access methods for relations and database rules can be maintained throughout the session. Ontobroker's API function dbaccess enables the generation of the instances from the given relational database on the fly.

9 Related Work

Deep annotation as we have presented it here is a cross-sectional enterprize.[12] Therefore there are a number of communities that have contributed towards reaching the objective of deep annotation. So far, we have identified communities for information integration (Section 9.1), mapping frameworks (Section 9.2), wrapper construction (Section 9.3), and annotation (Section 9.5).

9.1 Information Integration

The core idea of information integration lies in providing an algebra that may be used to translate information proper between different information structures. Underlying algebras are used to provide compositionality of translations as well as a sound basis for query optimization (cf., e.g., a commercial system as described in [20] with many references to previous work — much of the latter based on principal ideas issued in [31].

Unlike [20], our objective has not been the provisioning of a flexible, scalable integration platform *per se*. Rather, the purpose of deep annotation lies in providing a flexible framework for *creating the translation descriptions* that may then be exploited by an integration platform like EXIP (or Nimble, Tsimmis, Infomaster, Garlic, etc.). Thus, we have more in common with the approaches for creating mappings with the purpose of information integration described next.

9.2 Mapping and Merging Frameworks

Approaches for mapping and/or merging ontologies and/or database schemata may be distinguished mainly along the following three categories: discover, [22, 4, 6, 1, 19, 17], mapping representation [15, 1, 18, 21] and execution [5, 18].

In the overall area, closest to our own approach is [16], as it handles — like we do — the complete mapping process involving the three process steps just listed (in fact it also takes care of some more issues like evolution).

What makes deep annotation different from all these approaches is that for the initial discovery of overlaps between different ontologies/schemata they all depend on lexical agreement of part of

[12] Just like the Semantic Web overall!

the two ontologies/database schemata. Deep annotation only depends on the user understanding the presentation — the information within an information context — developed for him anyway. Concerning the mapping representation and execution, we follow a standard approach exploiting Datalog giving us many possibilities for investigating, adapting and executing mappings as described in Section 7.

9.3 Wrapper Construction

Methods for wrapper construction achieve many objectives that we pursue with our approach of deep annotation. They have been designed to allow for the construction of wrappers by explicit definition of HTML or XML queries [23] or by learning such definitions from examples [13, 3]. Thus, it has been possible to manually create metadata for a set of structurally similar Web pages. The wrapper approaches come with the advantage that they do not require cooperation by the owner of the database. However, their shortcoming is that the correct scraping of metadata is dependent to a large extent on data layout rather than on the structures underlying the data.

Furthermore, when definitions are given explicitly [23], the user must cope directly with querying by layout constraints and when definitions are learned, the user must annotate multiple web pages in order to derive correct definitions. Also, these approaches do not map to ontologies. They typically map to lower level representations, e.g. nested string lists in [23], from which the conceptual descriptions must be extracted, which is a non-trivial task. In fact, we have integrated a wrapper learning method, *viz.* Amilcare [3], into our OntoMat-Annotizer. How to bridge between wrapper construction and annotation is described in detail in [10].

9.4 Database Reverse Engineering

Database reverse engineering: There are very few approaches investigating the transformation of a relational model into an ontological model. The most similar approach to our approach is the project Infosleuth [11]. In this project an ontology is built based on the database schemas of the sources that should be accessed. The ontology is refined based on user queries. However, there are no techniques for creating axioms, which are a very important part of an ontology. Our approach is heavily based on the mapping of some database constraints into ontological axioms. Moreover, the semantic characteristics of the database schema are not always analyzed. More work has been addressed on the issue of explicitly defining semantics in database schemas [4], [16], extracting semantics out of database schema [4], [10] and transforming a relational model into an object-oriented model [3], which is close to an ontological theory. Rishe [16] introduces semantics of the database as a ”means” to closely capture the meaning of user information and to provide a concise, high-level description of that information. In [4] an interactive schema migration environment that provides a set of alternative schema mapping rules is proposed. In this approach, which is similar to our approach on the conceptual level, the re-engineer repeatedly chooses an adequate mapping rule for each schema artefact. However, this stepwise process creates an object-oriented schema, therefore axioms are not discussed.

9.5 Annotation

Finally, we need to consider information proper as part of deep annotation. There, we “inherit” the principal annotation mechanism for creating relational metadata as elaborated in [9]. The interested reader finds an elaborate comparison of annotation techniques there as well as in a forthcoming book on annotation [11].

10 Conclusion

In this deliverable we have described *deep annotation*, an original framework to provide semantic annotation for large sets of data. Deep annotation leaves semantic data where it can be handled best, *viz.* in database systems. Thus, deep annotation provides a means for mapping and re-using dynamic data in the Semantic Web with tools that are comparatively simple and intuitive to use.

To attain this objective we have defined a deep annotation process and the appropriate architecture. We have incorporated the means for server-side markup that allows the user to define semantic mappings by using OntoMat-Annotizer to create Web presentation-based annotations[13] and OntoMat-Reverse to create schema-based annotations. An ontology and mapping editor and an inference engine are then used to investigate and exploit the resulting descriptions either for querying the database content or to materialize the content into RDF files. In total, we have provided a complete framework and a demonstrator implementation for deep annotation.

For the future, there is a long list of open issues concerning deep annotation — from the more mundane, though important, ones (top) to far-reaching ones (bottom):

1. Granularity: So far we have only considered atomic database fields. For instance, one may find a string "Proceedings of the Eleventh International World Wide Web Conference, WWW2002, Honolulu, Hawaii, USA, 7-11 May 2002." as the title of a book whereas one might rather be interested in separating this field into title, location and date.

2. Automatic derivation of server-side web page markup: A content management system like Zope could provide the means for automatically deriving server-side web page markup for deep annotation. Thus, the database provider could be freed from *any* workload, while allowing for participation in the Semantic Web. Some steps in this direction are currently being pursued in the KAON CMS, which is based on Zope[14].

3. Other information structures: For now, we have built our deep annotation process on SQL and relational databases. Future schemes could exploit XQuery[15] or an ontology-based query language.

4. Interlinkage: In the future deep annotations may even link to each other, creating a dynamic interconnected Semantic Web that allows translation between different servers.

5. Opening the possibility to directly query the database, certainly creates problems such as new possibilities for denial of service attacks. In fact, queries, e.g. ones that involve too many joins over large tables, may prove hazardous. Nevertheless, we see this rather as a challenge to be solved by clever schemes for CPU processing time (with the possibility that queries are not answered because the time allotted for one query to one user is up) than for a complete "no go".

We believe that these options make *deep annotation* a rather intriguing scheme on which a considerable part of the Semantic Web might be built.

[13] The methodology "CREAM" and its implementation "OntoMat-Annotizer" have been intensively tested by authors of ISWC-2002 when annotating the summary pages of their papers with RDF metadata; see http://annotation.semanticweb.org/iswc/documents.html.

[14] see http://kaon.aifb.uni-karlsruhe.de/Members/rvo/kaon_portal

[15] http://www.w3.org/TR/xquery/

Acknowledgements.

We gratefully thank Leo Meyer, Dirk Wenke (Ontoprise), Gert Pache, and our colleagues at AIFB and FZI, for discussions and implementations that contributed toward the OntoLift demonstrator described in this deliverable.

References

[1] S. Bergamaschi, S. Castano, D. Beneventano, and M. Vincini. Semantic Integration of Heterogeneous Information Sources. In *Special Issue on Intelligent Information Integration, Data & Knowledge Engineering*, volume 36, pages 215–249. Elsevier Science B.V., 2001.

[2] J. Broekstra, A. Kampman, and F. van Harmelen. *Sesame: An Architecture for Storing and Querying RDF Data and Schema Information*. MIT Press, 2001.

[3] F. Ciravegna. Adaptive Information Extraction from Text by Rule Induction and Generalisation. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 1251–1256, San Francisco, CA, August 4–10 2001. Morgan Kaufmann Publishers, Inc.

[4] W. Cohen. The WHIRL Approach to Data Integration. *IEEE Intelligent Systems*, pages 1320–1324, 1998.

[5] T. Critchlow, M. Ganesh, and R. Musick. Automatic Generation of Warehouse Mediators Using an Ontology Engine. In *Proceedings of the 5 th International Workshop on Knowledge Representation meets Databases (KRDB'98)*, pages 8.1–8.8, 1998.

[6] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to map between ontologies on the semantic web. In *Proceedings of the World-Wide Web Conference (WWW-2002)*, pages 662–673. ACM Press, 2002.

[7] D. Fensel, J. Angele, S. Decker, M. Erdmann, H.-P. Schnurr, S. Staab, R. Studer, and Andreas Witt. On2broker: Semantic-based access to information sources at the WWW. In *Proceedings of the World Conference on the WWW and Internet (WebNet 99), Honolulu, Hawaii, USA*, pages 366–371, 1999.

[8] J. Golbeck, M. Grove, B. Parsia, A. Kalyanpur, and J. Hendler. New Tools for the Semantic Web. In *Proceedings of EKAW 2002*, LNCS 2473, pages 392–400. Springer, 2002.

[9] S. Handschuh and S. Staab. Authoring and Annotation of Web Pages in CREAM. In *Proceedings of the 11th International World Wide Web Conference, WWW 2002, Honolulu, Hawaii, May 7-11, 2002*, pages 462–473. ACM Press, 2002.

[10] S. Handschuh, S. Staab, and F. Ciravegna. S-CREAM — Semi-automatic CREAtion of Metadata. In *Proceedings of EKAW 2002*, LNCS, pages 358–372, 2002.

[11] Siegfried Handschuh and Steffen Staab, editors. *Annotation in the Semantic Web*, volume 96 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.

[12] Siegfried Handschuh, Steffen Staab, and Raphael Volz. On deep annotation. In *Proc. of the WWW-2003, Budapest, Hungary, May 2003*, pages 431–438. ACM, 2003.

[13] Nicholas Kushmerick. Wrapper Induction: Efficiency and Expressiveness. *Artificial Intelligence*, 118(1–2):15–68, 2000.

[14] V. A. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966. Russian original (1965) Doklady Akademii Nauk SSR 163-4 pp. 845-848.

[15] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *Proceedings of the 27th International Conferences on Very Large Databases*, pages 49–58, 2001.

[16] A. Maedche, B. Motik, N. Silva, and R. Volz. MAFRA - A Mapping Framework for Distributed Ontologies. In *Proceedings of EKAW 2002*, LNCS 2473, pages 235–250. Springer, 2002.

[17] D. McGuinness, R. Fikes, J. Rice, and S. Wilder. The Chimaera Ontology Environment. In *Proc. of AAAI-2000*, pages 1123–1124, 2000.

[18] P. Mitra, G. Wiederhold, and M. Kersten. A graph-oriented model for articulation of ontology interdependencies. In *Proceedings of Conference on Extending Database Technology (EDBT 2000)*. Konstanz, Germany, 2000.

[19] N. F. Noy and M. A. Musen. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *Proc. of AAAI-2000*, pages 450–455, 2000.

[20] Y. Papakonstantinou and V. Vassalos. Architecture and Implementation of an XQuery-based Information Integration Platform. *IEEE Data Engineering Bulletin*, 25(1):18–26, 2002.

[21] J. Y. Park, J. H. Gennari, and M. A. Musen. Mappings for Reuse in Knowledge-based Systems. In *Technical Report, SMI-97-0697, Stanford University*, 1997.

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

[23] A. Sahuguet and F. Azavant. Building intelligent Web applications using lightweight wrappers. *Data and Knowledge Engineering*, 3(36):283–316, 2001.

[24] S. Schulze-Kremer. Adding semantics to genome databases: Towards an ontology for molecular biology. In *5th Int. Conf. on Intelligent Systems for Molecular Biology*, Halkidiki, Greece.

[25] S. Staab, J. Angele, S. Decker, M. Erdmann, A. Hotho, A. Maedche, H.-P. Schnurr, R. Studer, and Y. Sure. Semantic Community Web Portals. *Proceedings of WWW9 / Computer Networks*, 33(1-6):473–491, 2000.

[26] N. Stojanovic, A. Maedche, S. Staab, R. Studer, and Y. Sure. SEAL: a framework for developing SEmantic PortALs. In *Proceedings of K-CAP 2001*, pages 155–162. ACM Press, 2001.

[27] R. Studer, Y. Sure, and R. Volz. Managing User Focused Access to Distributed Knowledge. *Journal of Universal Computer Science (J.UCS)*, 8(6):662–672, 2002.

[28] Y. Sure, J. Angele, and S. Staab. Guiding Ontology Developement by Methodology and Inferencing. In K. Aberer and L. Liu, editors, *ODBASE-2002 — Ontologies, Databases and Applications of SEmantics. Irvine, CA, USA, Oct. 29-31, 2002*, LNCS, pages 1025–1222. Springer, 2002.

[29] M. Vargas-Vera, E. Motta, J. Domingue, M. Lanzoni, A. Stutt, and F. Ciravegna. MnM: Ontology Driven Semi-automatic and Automatic Support for Semantic Markup. In *Proceedings of EKAW 2002*, LNCS 2473, pages 379–391. Springer, 2002.

[30] R. Volz, S. Handschuh, S. Staab, L. Stojanovic, and N. Stojanovic. Unveiling the hidden bride: Deep Annotation for Mapping and Migrating Legacy Data to the Semantic Web. *Web Semantics*, 2(1), 2004.

[31] G. Wiederhold. Intelligent integration of information. *Proceedings of the ACMSIGMOD International Conference on Management of Data*, pages 434–437, 1993.