Leopold-Franzens-University Innsbruck
Institute of Computer Science
Databases and Information Systems

Using Databases for Ontology
Processing, Storage and Reasoning
in Common and Mobile Environments

Master Thesis

supervised by:
Prof. Dr. Günther Specht

Wolfgang Gassler
Eva Zangerle

Innsbruck, August 27, 2007
Abstract

This master thesis describes the currently available approaches to store and reason about ontologies using databases. The Meta-Inn Approach, an enhanced version of the already existing approaches, is presented. It consists of an import engine, which was developed using XSLT and SQL/PL and maps the ontology information contained in an OWL file to a fast processable and optimized database format. The second part of the Meta-Inn implementation is the reasoning engine, which is able to derive new facts out of an existing ontology. It is implemented in SQL as well and uses common OWL inference rules. The second part of this document is concerned with the possibilities of using ontologies on mobile devices and its difficulties. The potential of an implementation of the Meta-Inn Approach using mobile databases and the associated problems are also analysed.
Acknowledgements

First of all we want to thank our supervisor and professor Günther Specht for giving us the opportunity to write this master thesis and for the support and advise throughout the last months.

Thanks to our friend Robert Binna, who set up the database server, kept it up and running and almost daily supported us in technical questions. Simon Bailey, who introduced us to the world of XSLT and provided us with his knowledge in these matters.

A big thanks goes to all our fellow students, with whom we enjoyed countless coffee breaks over the course of this thesis. Thank you to all our friends, who suffered from the fact that we spent more time writing on our master thesis than being with them. A very special thanks goes to our families, who have always supported us to realise our ideas and reach our goals.
Preface

The initial goal of this master thesis was to give a review on current international research projects concerning mobile databases and the processing of ontologies in mobile environments. Furthermore, missing parts or additional topics of the research area were to be discovered. Therefore we decided to browse all German universities manually and use search engines to find international projects dealing with this research topic. All well-known database companies were taken into account as well. The deflating result of the search process was that there was no single research project which covered the ontology processing on mobile devices using mobile databases. All found related papers are included in the bibliography. The found research projects are concerned with the processing and usage of ontologies for mobile applications [WGV06], however they are all based on a client-server architecture and do not use mobile databases or process the ontology locally on the mobile device. With no usable research projects for our purposes, we started our own research based on the paper of Weithöner, Liebig and Specht [WLS03]. The theoretical approach to port the current implementation to the mobile environment is described in chapters 4 and 5. After the theoretic idea we wanted to perform some tests to check whether the ideas can be realized by an implementation for mobile environments. Such an implementation only would have been possible if we had been able to build up the implementation on an already existing approach, such as the Meta Approach (chapter 4.2) or Oracle’s Approach (chapter 4.3). As the Meta Approach is only capable of processing RDF documents and the implementation is based on deductive databases, which are not available for mobile environments, it was not possible to build on top of it. The newer approach, developed by Oracle [DCES04], was not publicly available and also contact establishments with Oracle by email and telephone failed. Therefore we decided to implement a new approach to process ontologies using common database systems. For this reason the main practical goal of the thesis changed from the mobile environment back to the common environment. The implementation is called Meta-Inn Approach, as it is an extension of the Meta Approach and was developed in Innsbruck.
and is described in chapters 5 and 6. This approach extends the Meta Approach such that it is able to deal with OWL documents. Therefore a new mapping from OWL elements to database relations had to be defined. An new import and reasoning engine were implemented as well. Beside almost daily meetings to synchronize, discuss and merge our progress, the master thesis was split into the following parts, which were processed by the respective person.

Wolfgang Gassler
- Chapter 1 - Introduction
- Chapter 2 - Representation and Storage Concepts
- Chapter 3 - Reasoning and Querying
- Chapter 5 - Meta-Inn Approach (SQL part)
- Chapter 6 - Reasoning
- Chapter 8 - Ontologies in Mobile Environments
- Chapter 9 - Conclusion and Outlook
- Implementation of the database backend and the reasoning process
- Testing and Optimization using various example ontologies

Eva Zangerle
- Chapter 1 - Introduction
- Chapter 2 - Representation and Storage Concepts
- Chapter 4 - Ontology Management using Databases
- Chapter 5 - Meta-Inn Approach (XSLT part)
- Chapter 7 - Mobile Computing and Mobile Databases
- Chapter 8 - Ontologies in Mobile Environments
- Chapter 9 - Conclusion and Outlook
- Implementation of the import process using XSLT and SQL
- Testing and Optimization using various example ontologies

The tools and software used to realise this master thesis and the main parts of the implementation of the Meta-Inn Approach are listed in appendix A.1.
Contents

1 Introduction 1

2 Representation and Storage Concepts 3
  2.1 Introduction ........................................... 3
  2.2 RDF .................................................. 3
  2.3 OWL .................................................. 5
  2.4 Storage Concepts ................................... 7

3 Reasoning and Querying 8
  3.1 Query Languages ..................................... 9
  3.2 Reasoning Tools .............................. 9
    3.2.1 KAON 2 ......................................... 10
    3.2.2 Racer ........................................... 11
    3.2.3 FaCT / FaCT++ ................................ 11
    3.2.4 Pellet ........................................... 11
    3.2.5 Jena ............................................. 12
    3.2.6 Ontobroker ..................................... 12

4 Ontology Management using Databases 13
  4.1 Direct Approach ...................................... 13
  4.2 Meta Approach ....................................... 15
  4.3 Oracle OWL Approach ................................ 18

5 Meta-Inn Approach 22
  5.1 Motivation ........................................... 22
  5.2 Mappings ............................................. 23
  5.3 Import and Implementation ...................... 33

6 Reasoning 41
  6.1 Implementation ..................................... 45

7 Mobile Computing and Mobile Databases 50
  7.1 Mobile Databases ................................. 51
    7.1.1 Virtual-Primary-Copy ....................... 52

VII
## CONTENTS

7.1.2  Snapshot Strategies ........................................ 52  
7.1.3  Replication Proxy Servers .................................. 52  
7.1.4  Synchronization Markup Language (SyncML) ... 53  

8  Ontologies in Mobile Environments 55  
8.1  Ontologies in mobile Environments using mobile databases 56  
8.2  Merging .............................................................. 61  

9  Conclusion and Outlook 65  

Appendix 67  
A.1  Used Software and Tools ........................................ 67  
A.2  Database Schema .................................................. 67  
A.3  XSLT ................................................................. 70  
A.4  SQL Procedures - Documentation .............................. 79  
A.5  User Defined Functions - Documentation ..................... 84  
A.6  Example Ontology .................................................. 88  

Bibliography 97
Chapter 1

Introduction

The importance and usefulness of ontologies are undisputed and are shown in many fields of application. One of these fields is the area of tourism, in which the international communication between all stakeholders is simplified by using ontologies. One of the biggest problem in the field of CRS (Computer Reservation System) and GDS (Global Distribution System) is to wrap up all involved, heterogeneous computer systems and users. Especially the glossary of tourism is very large, ambiguous and depends on the location and country. To handle this issue and be able process all data homogeneously, the use of ontologies is an obvious choice. One of these approaches in this research area is the Harmo-TEN project \(^1\), which built up an ontology for accommodation, attractions, sights, events and restaurants. This data can be used in already existing systems to provide a common standardized language without changing the existing data or message formats in the communication process.

The creation of common glossaries is one of the main usage areas of ontologies and can be found in many other fields of application, such as medicine and e-commerce. One of the biggest ontology databases is the Gene Ontology project \(^2\), which consists of 26,000 term definitions. In the area of e-commerce, especially in the field of B2B, ontologies simplify the communication process between the business partners by providing a common language concerning catalogue management and sales chains. Especially upcoming fields such as personalization and context-awareness can be realized more efficiently and flexible by using ontologies [WGV06]. As the mobile field of application is developing very fast, these fields are also en route to the mobile environment.

In mobile environments the client-server concept currently is used, where the server has to cope with computation workload and the client is only

---

\(^1\)http://www.harmo-ten.org/
\(^2\)http://www.geneontology.org/
CHAPTER 1. INTRODUCTION

responsible for the presentation of information. The reason for this fact is that complex reasoning computations, which derive new facts out of an existing ontology, can only be done on the server side in an appropriate amount of time. During the last years the power of mobile devices increased heavily, which leads to the idea of splitting the workload and move simple computations to the client-side. Especially for servers, which are serving a huge number of clients, even a small shift of the workload to the client-side could decrease the server load significantly. The shifting of computation tasks to the client-side allows to give up the “not-always-on” constraint, which is elementary in the client-server concept. Additionally the connection is relieved, as the server only has to send the data once and the client can operate on this data independently. One possibility to efficiently use ontologies is to store them in a database.

This master thesis mainly deals with the problem of storing ontology data in a database. Firstly three already existent approaches of storing ontologies in databases are analysed. The Direct Approach was the first attempt to store ontologies in a database and turned out not to be as performant as the Meta Approach. The Meta Approach maps the ontology data to the database relations in an improved way. The third approach was developed by Oracle and is similar to the Meta Approach. Based on the results of the analysis of these three approaches, the Meta-Inn approach is developed. This approach extends the Meta Approach such that also ontologies, which are saved in an OWL file, can be stored in databases using an optimized format. It is implemented using XSLT and stored procedures and supports the reasoning about ontologies. The reasoning engine is able to derive new facts out of the imported ontology information and provides a simple access method to the inferred data by using SQL.

This paper is organized as follows. In the next chapter basic underlying storage technologies, such as RDF and OWL, are explained. Chapter 3 presents the most popular ontology reasoning tools, which can be used to process ontology data and derive new facts out of the imported ontology information. Chapter 4 is concerned with the existing approaches for storage and reasoning based on databases. Chapter 5 describes the new Meta-Inn Approach for the storage of ontologies in databases. The next chapter deals with reasoning based on the Meta-Inn Approach. Subsequently, mobile databases are introduced in chapter 7. The last chapter is concerned with the application of ontologies in mobile environments and the adaptation of the Meta-Inn Approach to mobile environments using mobile databases.
Chapter 2

Representation and Storage Concepts

2.1 Introduction

Due to the ever-increasing demand for ontologies, the need for a standardized interchange format emerged. The possibility to use XML was obvious, as XML is a common and flexible standard language. Basically there are two main approaches to represent an ontology, which are explained in the following.

2.2 RDF

The first approach was to use the existing RDF standard for the storage of ontologies. RDF stands for Resource Description Framework. It is a XML-based W3C\(^1\) recommendation to describe physical objects, abstract concepts or any other resource, which can be identified. As RDF was designed to describe resources on the net, an URI (Uniform Resource Identifier) is used to identify the resource. Its properties are described by a property name and a property value. The resulting triple consisting of an URI (subject), a property name and a property value (object) can completely describe any resource.

The example below shows two triples describing the website identified by the URI \texttt{http://dbis-informatik.uibk.ac.at}. The Dublin core [DCM00] standardized properties “title“ and ”description“ and the respective values complete the triples.

\footnote{\texttt{http://www.w3c.org}}
Listing 2.1: RDF Triples

As RDF does not support the declaration of relationships, the need for a more powerful ontology description language emerged. This is the reason why RDFS (Resource Description Framework Schema) is more suitable to describe ontologies. Additional features such as a system of classes and subclasses, properties and subproperties, domains, ranges and data types help to structure and link the resources.

The concept of classes and subclasses provides the opportunity of grouping several resources holding the same properties to one class. A class itself is represented as a resource and can be handled in the same way. In order to extend a class, the subClassOf property can be used to define a class that inherits all properties of the superclass. The same holds for the property and subproperty relationship. The possibility of creating subproperties leads to a hierarchical structure of properties and increases the machine-processibility.

The definition of range and domain restricts the possible classes of property values. The object can be restricted by the range attribute and the subject is restricted by the domain attribute, as shown in the following example.

Listing 2.2: RDF Domain Range

Some additional built-in data structures help to organize similar data. The simplest data structure is a container, which either orders its content or not. A more complex data structure is a collection, which provides two
additional features. Firstly it is possible to close a collection, secondly the content of a collection can be represented as a list.

RDF was the first attempt to store ontologies and therefore lacks some important features, which have been added by newer, mostly RDF-based formats.

2.3 OWL

The expressive power of RDF / RDFS proved to be very limited. Especially the lack of being able to define logical combinations of ontologies lead to an extension of RDFS. OWL (Web Ontology Language) is a combination of DAML-ONT (American approach) and OIL (European approach). The most important extensions include [SS04, WWW04]:

- Local scope of properties: scope restriction of rdfs:range property to certain classes.
- Disjointness of classes: two classes can be defined to be disjoint (e.g. male and female).
- Boolean combinations of classes: ability to create new classes using union, intersection and complement of already existing classes.
- Cardinality restriction: defines the minimum, maximum or exact number of distinct property values.
- Property characteristics: a property can for instance be set to be the inverse of another property (e.g. "userOf" and "usedBy") , symmetric or transitive (e.g. if Innsbruck isLocatedIn Austria and Austria isLocatedIn Europe, then Innsbruck IsLocatedIn Europe).

The possibility to reason about ontologies allows automated reasoners to conclude significantly more implicit data and therefore extend the amount of information contained in an ontology. Further details about reasoning processes can be found in chapter 3.

OWL consists of three different sublanguages to maintain the balance between expressiveness and reasoning support in finite time. OWL Full implements all requirements regarding expressiveness, but has become too powerful to be decidable. This sublanguage is also fully upward compatible to RDF and therefore any legal RDF documents also is a valid OWL document. OWL DL (Description Logic) is a sublanguage of OWL Full. It contains all OWL Full language constructs, but their
CHAPTER 2. REPRESENTATION AND STORAGE CONCEPTS

use is restricted. This fact leads to better reasoning support. The dis-
advantage is that RDF and OWL are no longer fully compatible. OWL
Lite is a subset of OWL DL and excludes a certain set of constructors.
The handling of OWL Lite is easier and less complex, although the ex-
pressivity is restricted. These relations between the respective OWL
subsets can be summarized as follows:

- Every legal OWL Lite ontology is a legal OWL DL ontology.
- Every legal OWL DL ontology is a legal OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

Figure 2.1: Animal Example Ontology

Figure 2.1 and the OWL listing in appendix A.6 show the main part of
the OWL example ontology, which is reused in all further chapters to
explain the main concepts of this master thesis. The ontology consists of animals, its habitats and some relations to demonstrate the most important OWL concepts.

2.4 Storage Concepts

To avoid the repeated parsing of the XML documents before being able to process the data, the ontologies are stored in high-performance data structures. The simplest way to store an ontology is to store the subjective, objective und predicate tripels, which can be extracted directly from the XML document. A second approach is to store the ontology as a graph. The storage using databases is described in detail in chapter 4.

In this chapter two ways of representing an ontology were represented. There are other formats such as Topic Maps, which in practice play a minor role. Both XML-based formats are very popular, but as OWL has a much higher expressive power than RDF, OWL is the leading language for the representation of ontologies.
Chapter 3

Reasoning and Querying

Beside being able to define ontologies, further tools for the efficient and practical use of ontologies are needed. The computability power of a computer enables to inspect all defined facts and offers the possibility to derive new facts based on the specified facts of ontologies. If an ontology defines the class Clownfish as a subclass of Fish and Fish is defined as a subclass of Animal, the fact that a Clownfish is an Animal can be derived by a reasoning engine. This transitive relation is a very simple example, but also more complex relations can be encountered by defining more complex rules. For example the fact "Student S attends a Lecture of Professor P" can lead to the knowledge "Student S knows Professor P" by applying an according rule.

An automatic consistency checking of an ontology ensures the consistency of the manually created definitions and reveals common faulty insertions by the user. Reasoning engines are also able to detect contradictions within the various facts, which is hardly possible to do by hand, because even small ontologies contain a large number of derived facts.

Beside the reasoning process, a common, well-defined interface to access the definitions and derived facts is needed. This is accomplished through standardized query languages, which have the advantage of a higher abstraction level. Furthermore, query languages are software independent and domain specific (DSL, Domain Specific Languages), which means that they are optimized for certain domains (e.g. ontology environment).

Most reasoning tools are restricted to a specific query language. In the following section the most popular tools and query languages are described.
CHAPTER 3. REASONING AND QUERYING

3.1 Query Languages

Basically query languages can be categorized by the storage format of the ontology on which the queries are performed. As query languages are used to access ontologies, they are based on the OWL, RDFS or DAML / OIL format. The ontology query language syntax is derived from other popular query languages (e.g. SQL) or functional and logic programming languages (e.g. Lisp). Depending on the language, the expressive power of the queries may differ. Especially some features of the ontology domain are not supported by all languages. SPARQL\(^1\) is the most widely-used query language for querying RDF data and is supported by most of the reasoning tools described in the next section.

3.2 Reasoning Tools

Reasoning tools are packages or programs which provide persistent storage and a standardized interface to manage and access ontologies. The DIG Interface developed by the DL Implementation Group\(^2\) is one of these standardized interfaces. It enables the communication with Description Logic Systems based on XML. As OWL currently is the most popular ontology language, modern reasoning tools work on the OWL standard. To be able to make reasoning decidable, most tools use OWL-DL or a subset of it. In the following section the most popular OWL based reasoners are described in detail.

The description logic expressivity of the reasoning tools is denoted as follows:

\[ \mathcal{AL} \]

Attributive language. This is the base language which allows:

- Atomic negation (negation of concepts that do not appear on the left hand side of axioms)
- Concept intersection
- Universal restrictions
- Limited existential quantification

\(^1\)http://www.w3.org/TR/rdf-sparql-query/  
\(^2\)http://dl.kr.org/dig/
\( \mathcal{FL}^- \) A sublanguage of \( \mathcal{AL} \), which is obtained by disallowing atomic negation.

\( \mathcal{FL}_o \) A sublanguage of \( \mathcal{FL}^- \), which is obtained by disallowing limited existential quantification.

\( \mathcal{C} \) Complex concept negation.

\( \mathcal{S} \) An abbreviation for \( \mathcal{AL} \) and \( \mathcal{C} \) with transitive properties.

\( \mathcal{H} \) Role hierarchy (subproperties - rdfs:subPropertyOf).

\( \mathcal{R} \) Limited complex role inclusion axioms; reflexivity and irreflexivity; role disjointness.

\( \mathcal{O} \) Nominals. (Enumerated classes of object value restrictions - owl:oneOf, owl:hasValue).

\( \mathcal{I} \) Inverse properties.

\( \mathcal{N} \) Cardinality restrictions (owl:Cardinality, owl:MaxCardinality).

\( \mathcal{Q} \) Qualified cardinality restrictions (available in OWL 1.1, cardinality restrictions that have fillers other than owl:thing).

\( \mathcal{F} \) Functional properties.

\( \mathcal{E} \) Full existential qualification (existential restrictions that have fillers other than owl:thing).

\( \mathcal{U} \) Concept union.

\( (D) \) Use of datatype properties, data values or data types.

Table 3.1: Description Logic Expressivity

3.2.1 KAON 2

KAON\(^3\) is one of the eldest reasoning systems developed by the University of Manchester and the University of Karlsruhe. It is currently released as KAON version 2, which is based on OWL-DL and F-Logics. The system supports the DL-safe subset of SWRL (Semantic Web Rule Language\(^4\), extension of OWL with Horn-like rules) with the expressivity

\(^3\)http://kaon2.semanticweb.org

\(^4\)http://www.w3.org/Submission/SWRL
CHAPTER 3. REASONING AND QUERYING

$SHIQ(D)$. This fact makes reasoning with KAON decidable. KAON does not use tableaux algorithms for the reasoning, as most of the other tools do. It uses new algorithms, which reduce a $SHIQ(D)$ knowledge base (ontology) to a disjunctive datalog program. This allows the program to reason about ontologies - as stated on the website - a magnitude faster than existing systems. As for the querying of ontologies, either SPARQL or F-Logic can be used as query languages.

3.2.2 Racer

Racer$^5$ is a commercial tool to reason and query large ontologies. The reasoning supports A-Boxes (concepts) and T-Boxes (individuals) and is done by an optimized tableaux calculus. Racer handles OWL-DL without nominals, therefore the expressivity class is $SHIQ(\text{ALCQHR}^+)$. There is also a basic implementation for SWRL syntax based rule language. It supports many interfaces such as DIG or APIs for Java and Lisp. Currently OWL-QL$^6$ and SPARQL (in development) can be used as query languages. The queries are converted to the new Racer Query Language (nRQL).

3.2.3 FaCT / FaCT++

FaCT and FaCT++$^7$ can only be used for reasoning about ontologies. Both tools use the tableaux algorithm for $SHOIQ$ description logics. FaCT++ (Fast Fast Classification of Terminologies) is a new implementation of FaCT, which was implemented in LISP. For performance reasons FaCT was newly implemented in C++, leading to the release of FaCT++.

3.2.4 Pellet

Pellet$^8$ is an open source reasoning tool, developed at the University of Maryland’s Mindswap Lab. It uses tableaux algorithms for description logics and supports the full expressivity of OWL-DL. It is written in Java and supports the description logic expressivity $SHOIN(D)$ and $SROIN(D)$ with OWL 1.1. The reasoning engine consists of the following parts:

- Consistency checking: finds all contradictory facts
- Concept satisfiability: checks the satisfiability of all classes, checks if instances are possible

---

$^5$http://www.racer-systems.com/
$^6$http://www-ksl.stanford.edu/projects/owl-ql/
$^7$http://owl.man.ac.uk/factplusplus/
$^8$http://pellet.owldl.com/
CHAPTER 3. REASONING AND QUERYING

• Classification: evaluates all subclass relations and builds the class hierarchy of all classes

• Realization: derives types of all possible individuals according to the class hierarchy

Pellet offers an analyse and repair module, which analyses any arbitrary OWL ontology and tries to convert it to an OWL DL document, even if it is an OWL Full document. Queries can be formulated in SPARQL, SWRL for rule definitions is partly supported.

3.2.5 Jena

Jena\(^9\) is an Open Source Java framework to build semantic web applications. It supports RDF, RDFS, OWL and SPARQL for querying. The system provides a flexible interface on which other inference modules or reasoners can be attached. The included rule-based OWL reasoner supports OWL-Lite. For more complex reasoning support the authors recommend the usage of full reasoners, such as Pellet or Racer connected by the Jena DIG interface.

3.2.6 Ontobroker

Ontobroker\(^{10}\) has originally been developed as a research prototype at the University of Karlsruhe and is now a commercial product of the company Ontoprise\(^{11}\). It supports OWL, RDF and F-Logic, consists of a Datalog reasoning engine and is able to use common database systems as persistence storage. To access the stored ontology data F-Logic, SPARQL and a SQL-like language can be used. As Ontobroker is developed by a company, there are hardly any information about the implementation or core of the system available.

\(^9\)http://jena.sourceforge.net
\(^{10}\)http://www.ontoprise.de/documents/datasheetontobroker.pdf
\(^{11}\)http://www.ontoprise.de

Eva Zangerle
Chapter 4

Ontology Management using Databases

In practice not only the creation and the reasoning about ontologies are a crucial factor, the field of storage is also very substantial. Especially the concepts of concurrent multiuser access, persistent storage, fault tolerance, consistence, standardized storage access methods and optimized disk allocation are important features for ontology processing systems. The use of databases as storage layer for ontologies would be an obvious choice, because databases are well developed and offer all these features in order to be able to store data in an efficient way. Databases provide a transactional system to handle concurrent users and guarantee the consistent storage of data even in case of a system failure. Additionally the storage is executed in an optimized way in regards of disk space usage, access time and can also include other concepts such as encryption or compression. In the following sections, the approaches using databases for the storage of ontologies are presented.

4.1 Direct Approach

The most intuitive approach is to use logical database systems to store the logical information of ontologies in a persistent way and be able to use inference engines to deduce new facts and rules from already existing information. As logical databases use declarative languages such as Datalog, it is necessary to transform ontologies into logic programs, which can further on be evaluated by these languages. This transformation was introduced by Grosof and Horrocks [GHVD03] in 2003 and is called ”Direct Approach“ in this thesis. As logical databases are not scalable, have representational disadvantages and are not state of the art anymore, the mapping can also be used to realize Datalog programs on top of relational databases. In this approach every class is mapped
to a unary relation and every property is mapped to a binary relation. Property constructor statements, such as a subclass relationship, are mapped to a set of rules. Rules are realized by relational views or SQL statements, which are operating on the views. Table 4.1 shows OWL statements and the respective rules resulting from the application of the Direct Approach.

<table>
<thead>
<tr>
<th>OWL Abstract Definition</th>
<th>DLP Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition of class c</td>
<td>Rule set Rc</td>
</tr>
<tr>
<td>SubClassOf(c b)</td>
<td>c(X) :- b(X)</td>
</tr>
<tr>
<td>SubClassOf(unionOf(b_1...b_n) c)</td>
<td>c(X) :- b_1(X)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>c(X) :- b_n(X)</td>
</tr>
<tr>
<td>SubClassOf(c intersectionOf(b_1...b_n))</td>
<td>c(X) :- b_1(X), ..., b_n(X)</td>
</tr>
<tr>
<td>SubClassOf(intersectionOf(b_1...b_n) c)</td>
<td>b_1(X) :- c(X)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>b_n(X) :- c(X)</td>
</tr>
<tr>
<td>SubClassOf(c restriction(p allValues-From(b)))</td>
<td>c(X) :- p(X, b), anonID(X)</td>
</tr>
<tr>
<td>SubClassOf(restriction(p someValues-From(b)) c))</td>
<td>anonID(X) :- p(X, b), c(X)</td>
</tr>
</tbody>
</table>

Table 4.1: Direct Approach Rules, [WLS03] p.8

Clownfish("Nemo")
Clownfish("Marlin")
livesIn("Clownfish", "Ocean")

In the above example two instances of Clownfish are initialized. To define that clownfish live in the ocean, a new relation "livesIn" has to be created. This relation can be used for further relationships of the same type, but every new property results in a new relation for all property instantiations.
CHAPTER 4. ONTOLOGY MANAGEMENT USING DATABASES

This approach was analysed by [WLS03] and showed some serious disadvantages:

- The class names cannot be accessed from inside the logic program because the class names are only stated in the relation name. Therefore it is not possible to determine all the classes of which an individual I is instance of. As the subclass relationships are mapped to respective rules, the only solution would be to go through all the classes manually and ask whether I is an instance of this class. However this approach would require the knowledge of all the class names.

- As every class and every property are mapped to a separate relation, this approach results in a huge amount of relations which only contain a few entries representing instantiations.

- The set of rules is varying for every ontology. If an ontology increases in the number of classes and concepts, the set of rules grows as well, as all concepts (e.g. subclass relation) are mapped to new rules.

- Not only the set of rules varies for every ontology, also the names of the relations and the structure of the rules differ for every ontology. Therefore no precompilations or optimizations are possible.

4.2 Meta Approach

The second approach, which is an optimized version of the Direct Approach, was introduced by Weithöner, Liebig and Specht [WLS03] and is called ”Meta Approach”. It basically contains two improvements which can be applied to the Direct Approach:

- A meta level is introduced, where the rules and facts are pushed onto. The names of concepts and properties are now arguments of ”meta predicates”. Like this, the problem of not being able to access the class name from inside the logic program is solved.

- To avoid the growth of the rule set for an ontology, a constant set of rules, which is valid for any ontology, is introduced. The names of the rules are the same for every ontology, therefore the names of the resulting relations are the same. This fact can be used for the improvement of database queries, because the database schema is fixed and known in advance.
The Meta Approach takes the facts resulting from the Direct Approach and pushes them onto a higher meta level according to the HiLog approach [CKW93]. In HiLog it is possible to position terms in places, where in First Order Logic only atomic formulas or predicates were allowed.

The storage is realized by two relations. The first one collects all class instantiations and is called type, whereas the second relation propInst contains all property instantiations (see table 4.2).

To define an instance Nemo of class Clownfish, a new entry is added to the binary relation type:

\[
\text{type}(\text{"Nemo", "Clownfish"})
\]

The property that Clownfish is the food of Shark can be expressed as an entry in the relation "propInst" as stated below.

\[
\text{propInst}(\text{"isFoodOf", "Clownfish", "Shark"})
\]

This relation takes three arguments, the constant name of the property (predicate), its subject and object. The two relations are sufficient to store all information without any loss and allow a direct access to all class names from within the logic program. Due to the Direct Approach the information about the relationship between Clownfish and Fish in the example gets lost. After applying the Direct Approach there exists a rule which defines that Nemo and all other Clownfish instantiations are also Fish, but the explicit knowledge about the relationship (SubClassOf) between the type Clownfish and the type Fish is no longer available.

<table>
<thead>
<tr>
<th></th>
<th>Direct Approach</th>
<th>Meta Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nemo is a</td>
<td>Clownfish(Nemo)</td>
<td>type(Nemo, Clownfish)</td>
</tr>
<tr>
<td>Clownfish</td>
<td>isFoodOf(Clownfish, Shark)</td>
<td>propInst(isFoodOf, Clownfish, Shark)</td>
</tr>
<tr>
<td>Clownfish is</td>
<td>food of Shark</td>
<td></td>
</tr>
<tr>
<td>Clownfish</td>
<td>isFoodOf(Clownfish, Shark)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of Direct and Meta Approach

After having stored the knowledge by using the Meta Approach, further information can be discovered by applying rules to the data. In the following example Clownfish is defined as a subclass of Fish. Nemo is of the type Clownfish and Fish is a subclass of Animal.
CHAPTER 4. ONTOLOGY MANAGEMENT USING DATABASES

\begin{verbatim}
isSub("Fish", "Animal")
isSub("Clownfish", "Fish")
type("Nemo", "Clownfish")
\end{verbatim}

From the above example one can conclude that Nemo also is a subclass of class Fish, which can be derived by the following rule:

\begin{verbatim}
type(I, X) :- isSub(Y, X), type(I, Y)
\end{verbatim}

The above rule defines that if an individual I is instance of class Y and Y is subclass of class X, I is also an individual of class X. It can be applied to the above example like that:

\begin{verbatim}
type("Nemo", "Fish") :-
isSub("Clownfish", "Fish"),
type("Nemo", "Clownfish")
\end{verbatim}

Additionally, the transitivity relationship can be defined as follows:

\begin{verbatim}
isSub(X, Y) :- isSub(X, Z), isSub(Z, Y)
\end{verbatim}

Applying the transitivity rule to the above example leads to the fact that the class Clownfish is also a subclass of Animal.

\begin{verbatim}
isSub("Clownfish", "Animal") :-
isSub("Clownfish", "Fish"),
isSub("Fish", "Animal")
\end{verbatim}

As can be seen in this example, the rules are independent of any instantiations and are suitable for any ontology. With the combination of the ontology specific facts and the general rules, all A- and T-Box queries can be performed. In table 4.3 various A- and T-Box queries for the Meta- and the Direct Approach are compared.

To process the specified queries, the Meta Approach only needs to create simple queries on stored knowledge, whereas the Direct Approach requires information about all class names, which are only available as the relation names and cannot be accessed from within the queries. The Meta Approach makes it possible to walk through the relations and answer hierarchical queries.

In order to be able to query and reason about the stored information the rule set of the intersection of OWL and LP (OWLP) has to be applied to the data. As the rule set of the Meta Approach is suitable for any ontology the implementation and querying can be optimized in advance.
Table 4.3: Comparison between Direct and Meta Approach, [WLS03]

<table>
<thead>
<tr>
<th>Query</th>
<th>Meta Approach</th>
<th>Direct Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is given individual i instance of given class C?</td>
<td>?type(&quot;i&quot;, &quot;C&quot;)</td>
<td>?C(&quot;i&quot;)</td>
</tr>
<tr>
<td>List all instances of given class C.</td>
<td>?type(I, &quot;C&quot;)</td>
<td>?C(I)</td>
</tr>
<tr>
<td>List all classes given individual i is instance of.</td>
<td>?type(&quot;i&quot;, &quot;C&quot;)</td>
<td>Manually go through every known class C and check for ?C(&quot;i&quot;).</td>
</tr>
<tr>
<td>Check if given class C is subclass of given class C.</td>
<td>?isSub(&quot;C&quot;, &quot;D&quot;)</td>
<td>Create new instance ic of class C and check whether ic is also instance of class D.</td>
</tr>
<tr>
<td>List subclass of given class C.</td>
<td>?isSub(X, &quot;C&quot;)</td>
<td>Manually go through every known class D create new instance id from this class. Check whether id is also instance of C.</td>
</tr>
</tbody>
</table>

One can see that the Meta Approach consists of many enhancements, as no knowledge get lost and the size of the rule set and relations keeps constant. The rules and relations can be used for all ontologies and can therefore be optimized in advance. The schema of the Meta Approach supports the reasoning process and simplifies it. Although the approach is not optimized for relational database systems and supports only RDF.

### 4.3 Oracle OWL Approach

A very similar approach to store ontologies in relational database management systems was introduced by Das, Chong, Eadon and Srinivasan of the Oracle corporation [DCES04]. It uses the Oracle database system...
and processes, in contrast to the Meta Approach, which only covers RDF based ontologies, OWL DL and OWL Lite based ontologies. Additionally this approach provides functions to create queries in an easy way and be able to access the knowledge in a higher abstraction level. The knowledge storage is realized in a similar way to the Meta Approach and stores the information in six relations without any loss of knowledge.

Ontologies (OntologyID, OntologyName, Owner, ...)
Terms (TermID, OntologyID, Term, Type, ...)
Properties (OntologyID, PropertyID, DomainClassID, RangeClassID, Characteristics, ...)
Restrictions (OntologyID, NewClassID, PropertyID, MinCardinality, MaxCardinality, SomeValuesFrom, AllValuesFrom, ...)
Relationships (OntologyID, TermID1, PropertyID, TermID2, ...)
PropertyValues (OntologyID, TermID, PropertyID, Value, ...)

Listing 4.1: Storage Relations

The Ontologies relation contains basic information about the various ontologies. The Terms relation defines all lexical representations (attribute Term) of classes, instantiations and properties (attribute Type), which are used in the ontologies. They can be referenced by the primary key TermID and can be used by multiple entities in different ontologies. In the relation Properties, information about the properties, e.g. domain and range are stored. The Characteristics - attribute specifies which properties, such as symmetry, transitivity, functional inverse or functional, are valid for the entry. The relation Restrictions defines restrictions on property values, which internally results in a new class containing the restricted properties. Relationships specifies relations between two terms. In the relation PropertyValues, <property, value> - keys are stored.

The relations also contain some bootstrap entries, which represent the basic OWL concepts. For example the relation "subClassOf" is stored as a transitive property relating two classes. This approach includes an inference system as well. It applies the OWL Lite and OWL DL basic rule set to the data and derives new properties which are stored
CHAPTER 4. ONTOLOGY MANAGEMENT USING DATABASES

Persistently and are available for further queries. Beside the storage of ontologies, this approach also includes methods to use the stored knowledge for further applications. The ontology knowledge therefore can be used to generate more flexible and intelligent queries on other contents. To create such queries, the stored knowledge is accessed by functions which provide a simplified interface. The following example shows the application of the \texttt{ONT\_RELATED} procedure. It detects whether two terms are related by a specified type of relation in a given ontology.

The relation ”MarineAnimals” in table 4.4 shows some marine animals.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nemo</td>
<td>Clownfish</td>
</tr>
<tr>
<td>Marlin</td>
<td>Clownfish</td>
</tr>
<tr>
<td>Flipper</td>
<td>Dolphin</td>
</tr>
<tr>
<td>White Shark</td>
<td>Shark</td>
</tr>
</tbody>
</table>

Table 4.4: Marine Animals

In the following query the semantic knowledge as defined in the example ontology is used to get all animals of type Fish, although the fact if the animal is a fish or not, is not defined in the relation MarineAnimals.

\begin{verbatim}
SELECT r.Name FROM MarineAnimals r
WHERE ONT\_RELATED(Type, 'IS\_A', 'Fish', 'example\_ontology')=1;
\end{verbatim}

Listing 4.2: Extended Query Example

As Clownfish and Shark are subclasses of Fish, Nemo, Marlin and White Shark are in the result set. Dolphin is not a Fish and therefore not present in the result set. In this example the relation type IS\_A is used, even though all relation types defined in the ontology can be used. The function also supports combinations of relation types such as IS\_A OR EQV, which detects all subclasses or classes equal to the specified class.

Internally the \texttt{ONT\_RELATED} procedure is realized by the transitive closure feature indicated by the SQL structure:
CHAPTER 4. ONTOLOGY MANAGEMENT USING DATABASES

SELECT ... 
FROM ... 
START WITH <start_condition> 
CONNECT BY <connect_condition>; 

Listing 4.3: SQL Transitive Closure Query Structure

The transformation of the statement from the example in listing 4.2 is stated below:

SELECT r.Name FROM MarineAnimals r 
WHERE r.Type IN 
 (SELECT term1 FROM relationships 
  START WITH 
   term2 = 'Fish' AND 
   relation = 'IS_A' 
  CONNECT BY 
   PRIOR term = term2 AND 
   relation = 'IS_A'); 

Listing 4.4: Transformed Query

The following list gives an overview of the available stored procedures to access the stored ontology knowledge:

- **ONT_RELATED** as described above
- **ONT_DISTANCE** calculates the distance of the shortest path to a related term detected by **ONT_RELATED**
- **ONT_PATH** returns the shortest path to a related term found by **ONT_RELATED**
- **ONT_EXPAND** returns a table which includes found terms, the relation type, the distance and the path

In this chapter the three main approaches for the storage of ontologies were presented. The Direct Approach was the first attempt to store ontologies in a database-processable format. However it showed some serious shortcomings, such as the loss of meta-information. Various improvements of the Direct Approach were introduced by the Meta Approach, which is able to map ontology data to a database without any loss of information by using an additional meta-level. Oracle developed an approach, which is also able to handle OWL documents and its expressivity. It can also be used to enhance existing queries on relational databases by using the knowledge of ontologies. All three approaches are based on the same concept, although only the third approach is currently able to fulfill the requirements of the market.

Wolfgang Gassler, Eva Zangerle
Chapter 5

Meta-Inn Approach

5.1 Motivation

As shown in the last chapter, the currently available approaches and implementations to process ontologies using databases have not reached the end of the optimization process. The Meta Approach is only feasible for RDF documents, which are mostly replaced by the new standard OWL. As OWL is very expressive and introduced many new features, the concept of the Meta Approach is no longer sufficient to handle all characteristics of modern ontology information. Furthermore, the Meta Approach was developed on deductive database systems, which have lost their popularity and are no longer widely used. Therefore one of the main parts of this master thesis is the extension of the Meta Approach \[WLS03\] to get rid of these disadvantages and adapt the concept to be able to run on modern relational database systems and handle the widely accepted format OWL to store ontologies. To advance the Meta Approach, elements and ideas of the Oracle OWL Approach \[DCES04\] are also incorporated in the optimization process to combine all existing ideas and concepts to maximise the gain of the new extended approach. Moreover one of the main focuses of the Meta-Inn Approach is the reasoning process. The common reasoning rules which derive new facts out of the imported OWL file are realized with SQL/PL and recursive SQL-statements. The new mapping and import process to convert an OWL file to a fast processable database format is also part of this master thesis. It is implemented in a very simple, but powerful way by using XSLT and database procedures.
5.2 Mappings

The core of the Meta-Inn Approach is the schema of the relational database to store the ontology information in a fast processable and optimised way. All further processes and computations, e.g. the reasoning process, can benefit from this optimised storage and access or manipulate the data directly by using SQL statements.

The core of the schema consists of the following three relations to store the information contained in an ontology.

\[
\begin{align*}
\text{Term} & \quad (\text{termID}, \text{termOntID}, \text{termName}, \text{termTypeID}) \\
\text{PropertyInst} & \quad (\text{propInstID}, \text{propID}, \text{propInstSub}, \text{propInstObj}) \\
\text{Restriction} & \quad (\text{restID}, \text{propID}, \text{restSub}, \text{restObj}, \text{restType}, \text{restCardinalityMin}, \text{restCardinalityMax})
\end{align*}
\]
CHAPTER 5. META-INN APPROACH

The relation Term contains all classes, properties, individuals and literals. Every term gets its own termID, which is unique over all saved ontologies and can therefore used for further references on terms. termOntID refers to the respective ontology and termTypeID indicates the type of the entry (e.g. Class, Property, Literal, etc.).

For example an OWL class `<owl:Class rdf:ID="Animal">` is mapped to the entry Term(termID, termOntID,"Animal", class). The relation PropertyInst stores all instantiations of properties which consist of three references to term entries. The first one references to the property, which should be instantiated. The two other references define the subject and object of the property. The subject refers to the parent element in the OWL tree and the object can either be another OWL element or a literal. In the second case the literal is stored as a new entry in the relation Term and referenced in the object column.

The mapping of the property instantiation of the property subClassOf is listed below.

```xml
<owl:Class rdf:ID="Fish">
  <rdfs:subClassOf rdf:resource="#Animal"/>
</owl:Class>
```

Meta-Inn Approach Mapping

PropertyInst(propInstID, subClassOf, Fish, Animal)

These two relations are very similar to the original Meta Approach and can nearly hold all OWL information, except for the structure of restrictions. To handle this more complex structure, the relation Restriction is introduced. As restrictions restrict the possible values of class properties, the same triple concept (property, subject, object) as described above is used to specify the possible values or instantiations. Furthermore, restrictions support the specification of cardinality, which is stored in the columns restCardinalityMin and restCardinalityMax. An exactly defined cardinality results in the same minimum and maximum value. Restrictions can be specified with the following three different types.

- hasValue
- someValuesFrom
- allValuesFrom

The type of a restriction is stored in the restType column and defines the semantic meaning of the restriction. The following example shows a
very simple restriction on the class AnimalInEurope, which defines that there must be at least one referenced habitat located in Europe.

```
<owl:Restriction>
  <owl:onProperty>
    <owl:ObjectProperty rdf:about="#livesIn"/>
  </owl:onProperty>
  <owl:someValuesFrom rdf:resource="#HabitatsInEurope"/>
  <owl:minCardinality>1</owl:minCardinality>
</owl:Restriction>
```

⇓ Meta-Inn Approach Mapping

Restriction(livesIn, AnimalsInEurope, HabitatsInEurope, someValuesFrom, 1, NULL)

These three relations Term, PropertyInst and Restriction suffice to store all information of an OWL specified ontology without any loss of information. Basically all OWL elements are processed and stored as described above, but there are a few exceptions, which need some workarounds or special handling to fit in the defined schema. OWL contains many derived classes and properties, e.g. DeprecatedClass, FunctionalProperty or DatatypeProperty, which are special cases of other classes and properties. As in the Term relation only class and property types are differentiated, the knowledge about the respective subtype of the class or property would be lost. Therefore an additional entry of type type is added to the PropertyInst relation and specifies the type of the class or property. Thus the property `<owl:FunctionalProperty rdf:ID="canSwim"/>` is mapped to the two following entries in the "Term" and "PropertyInst" relation:

```
Term (termID, termOntID, "canSwim", property)
PropertyInst (propInstID, type, canSwim, FunctionalProperty)
```

Elements with more than one object value, e.g. intersectionOf, unionOf or oneOf have to be handled differently as well. To cope with multiple object values, each object value is stored as a new entry in the PropertyInst relation. The following example shows the mapping of an unionOf structure with two object values.
CHAPTER 5. META-INN APPROACH

<owl:Class rdf:ID="AnimalFood">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#FoodInWater">
      <owl:Class rdf:about="#FoodOnLand">
    </owl:Class>
  </owl:Class>
</owl:Class>

⇓ Meta-Inn Approach Mapping

Term(AnimalFood, class)
PropertyInst(propInstID, subClassOf, Nemo, Clownfish)
PropertyInst(propInstID, livesIn, Nemo, Pacific)

Instances of user defined classes are stored in the Term relation with type inst. The class type of the instance is defined by a subClassOf relation and the property instantiations are saved in the PropertyInst relation as described above.

<Clownfish rdf:ID="Nemo">
  <livesIn rdf:resource="#Pacific"/>
</Clownfish>

⇓ Meta-Inn Approach Mapping

Term(termID, termOntID, "Nemo", inst)
PropertyInst(propInstID, subClassOf, Nemo, Clownfish)
PropertyInst(propInstID, livesIn, Nemo, Pacific)

Basically there are three ways of mapping the OWL elements to the database relations, which are described in the following part.

- If an individual, a new element or a new property is created, it is added to the Term - relation:
  
  Term(termID, termOntID, termName, termTypeID)

  The termID is an automatically incremented, unique ID for the term. termOntID specifies the currently used ontology, which is dropped in the following listing to simplify matters as it stays the same throughout the import session. termName contains the name of the term and termTypeID specifies the type of the term (e.g. class, property, etc.).

- If a property gets instantiated, a tuple is added to the PropertyInst - relation:
CHAPTER 5. META-INN APPROACH

PropertyInst(propInstID, propID, propInstSubject, propInstObject)

propInstID is an automatically incremented, unique ID for the property instantiation. propID stands for an entry in the table term, which defines the name of the property, e.g. owl:subClassOf. The property instantiation belongs to the propInstSubject value, which in most cases is the parent OWL element. propInstObject contains the value of the property entry.

- A restriction is mapped to the relations as follows:

Restriction(property, subject, object, restrictionType, minCard, maxCard)

property stands for the propertyID, with which the restriction is concerned. The restriction is set on the specified subject (mostly the parent OWL element) and restricts the value of the property to the specified object. type stands for the type of the restriction, it can be hasValue, someValuesFrom or allValuesFrom. minCard and maxCard are concerned with the minimal and maximal cardinality of the restriction, both values have to be numeric (or NULL).

Below, all possible mappings of RDF, RDFS and OWL elements are listed. For the sake of clarity, the automatically incremented key values are not shown in the listing. The underlined values are set directly by the respective element or its attributes. Parameters that are not directly concerned with the respective mapping, are not underlined and use the following abbreviations:

<table>
<thead>
<tr>
<th>T</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI</td>
<td>PropInst</td>
</tr>
<tr>
<td>R</td>
<td>Restriction</td>
</tr>
<tr>
<td>prop</td>
<td>Property</td>
</tr>
<tr>
<td>subj</td>
<td>Subject</td>
</tr>
<tr>
<td>obj</td>
<td>Object</td>
</tr>
<tr>
<td>min</td>
<td>minCardinality</td>
</tr>
<tr>
<td>max</td>
<td>maxCardinality</td>
</tr>
</tbody>
</table>

Table 5.1: Abbreviations
owl:allValuesFrom
⇓
Meta-Inn Approach Mapping
R(prop, subj, obj, allValuesFrom, min, max)

owl:backwardCompatibleWith
⇓
Meta-Inn Approach Mapping
PI(backwardCompatibleWith, subject, object)

owl:cardinality
⇓
Meta-Inn Approach Mapping
R(prop, subj, obj, type, cardinality, cardinality)

owl:Class
⇓
Meta-Inn Approach Mapping
T(ontology, className, class)

owl:complementOf
⇓
Meta-Inn Approach Mapping
PI(complementOf, subj, obj)

owl:DatatypeProperty
⇓
Meta-Inn Approach Mapping
T(propertyName, prop)
PI(type, propertyName, DatatypeProperty)

owl:DeprecatedClass
⇓
Meta-Inn Approach Mapping
T(ontology, className, class)
PI(subClassOf, className, DeprecatedClass)

owl:DeprecatedProperty
⇓
Meta-Inn Approach Mapping
T(propertyName, prop)
PI(type, propertyName, deprecatedProperty)
CHAPTER 5. META-INN APPROACH

owl:DataRange
⇓ Meta-Inn Approach Mapping
PI(DataRange, subj, obj1)
PI(DataRange, subj, obj2)
PI(DataRange, subj, objn)

owl:differentFrom
⇓ Meta-Inn Approach Mapping
PI(differentFrom, subj, obj)

owl:disjointWith
⇓ Meta-Inn Approach Mapping
PI(disjointWith, subj, obj)

owl:equivalentClass
⇓ Meta-Inn Approach Mapping
PI(equivalentClass, subj, obj)

owl:equivalentProperty
⇓ Meta-Inn Approach Mapping
T(propertyName, prop)
PI(type, propertyName, equivalentProperty)

owl:FunctionalProperty
⇓ Meta-Inn Approach Mapping
T(propertyName, prop)
PI(type, propertyName, FunctionalProperty)

owl:hasValue
⇓ Meta-Inn Approach Mapping
R(prop, subj, obj, hasValue, min, max)
CHAPTER 5. META-INN APPROACH

owl:imports
⇓ Meta-Inn Approach Mapping
PI(imports, ontology, literal)

owl:incompatibleWith
⇓ Meta-Inn Approach Mapping
PI(incompatibleWith, ontology, literal)

owl:intersectionOf
⇓ Meta-Inn Approach Mapping
PI(intersectionOf, subj, obj1)
PI(intersectionOf, subj, obj2)
PI(intersectionOf, subj, objn)

owl:InverseFunctionalProperty
⇓ Meta-Inn Approach Mapping
T(propertyName, prop)
PI(type, propertyName, InverseFunctionalProperty)

owl:inverseOf
⇓ Meta-Inn Approach Mapping
PI(inverseOf, subj, obj)

owl:maxCardinality
⇓ Meta-Inn Approach Mapping
R(prop, subj, obj, type, min, maxCardinality)

owl:minCardinality
⇓ Meta-Inn Approach Mapping
R(prop, subj, obj, type, minCardinality, max)
owl:oneOf
⇓ Meta-Inn Approach Mapping
PI(oneOf, subj, obj1)
PI(oneOf, subj, obj2)
PI(oneOf, subj, objn)

owl:onProperty
⇓ Meta-Inn Approach Mapping
R(propertyName, subj, obj, type, min, max)

owl:Ontology
⇓ Meta-Inn Approach Mapping
T(ontologyName, ontology)

owl:priorVersion
⇓ Meta-Inn Approach Mapping
PI(priorVersion, subj, obj)

owl:Restriction
⇓ Meta-Inn Approach Mapping
R(prop, subj, obj, type, min, max)

owl:sameAs
⇓ Meta-Inn Approach Mapping
PI(sameAs, subj, obj)

owl:someValuesFrom
⇓ Meta-Inn Approach Mapping
R(prop, subj, obj, someValuesFrom, min, max)

owl:SymmetricProperty
⇓ Meta-Inn Approach Mapping
T(propertyName, prop)
PI(type, propertyName, SymmetricProperty)
CHAPTER 5. META-INN APPROACH

owl:TransitiveProperty

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ T(\text{propertyName, prop}) \]
\[ PI(\text{type, propertyName, TransitiveProperty}) \]

owl:unionOf

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ PI(\text{unionOf, subj, obj1}) \]
\[ PI(\text{unionOf, subj, obj2}) \]
\[ PI(\text{unionOf, subj, objn}) \]

owl:versionInfo

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ PI(\text{versionInfo, subj, obj}) \]

rdf:type

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ PI(\text{type, subj, obj}) \]

rdfs:comment

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ PI(\text{comment, subj, obj}) \]

rdfs:domain

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ PI(\text{domain, subj, obj}) \]

rdfs:label

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ PI(\text{label, subj, obj}) \]

rdfs:isDefinedBy

\[ \downarrow \text{Meta-Inn Approach Mapping} \]

\[ PI(\text{isDefinedBy, subj, obj}) \]
5.3 Import and Implementation

In the following section the implementation of the import process is presented. The import process reads an OWL document and imports the extracted ontology information into the fast processable database format (see section 5.2). The implementation was developed from scratch and uses XSLT and already available technologies provided by the database system. As XSLT was designed for the manipulation of XML structures, it is particularly suitable to extract the ontology information from OWL files. All other necessary processing features, which exceed the power of XSLT, are realized with SQL/PL. Thus no other programming language is required and the only requirements are a XSLT processor and a relational database system.

XSLT, the Extensible Stylesheet Language for Transformation\(^1\), is an official recommendation of the World Wide Web Consortium (W3C) and provides a flexible and powerful way to transform XML documents into some other format. As the transformation is controlled by user defined rules, the output format is not limited by any means. Common output formats are XML, HTML, PDF but even Java code or - as in

\(^1\)http://www.w3.org/TR/xslt

Wolfgang Gassler, Eva Zangerle
CHAPTER 5. META-INN APPROACH

this implementation - SQL.

XSLT is based on pattern matching rules, so-called templates, which define how matched parts are converted to the output format. The XSLT language is very influenced by the design of functional programming and is therefore free of side-effects. Hence variables can not be changed and loop structures are realized by recursion. If all templates are side-effect free, the simultaneous execution of templates is possible and accelerates the processing.

Search patterns, on which templates match, are defined in XPath\(^2\), a standard to describe or address elements or parts of XML structures. Listing 5.1 shows the template used to match the OWL element <owl:Ontology>. The template creates the SQL procedure call CALL create_ontology and specifies the value of the rdf:about attribute of the owl:Ontology element as its argument.

\[
\begin{align*}
\text{\texttt{<xsl:template match="owl:Ontology">}} \\
\text{\texttt{CALL create_ontology(}} \\
\text{\texttt{\quad \texttt{'<xsl:value-of select="@rdf:about"/>'}});}} \\
\text{\texttt{\texttt{\textless/xsl:template}>}}
\end{align*}
\]

Listing 5.1: XSLT template to match owl:Ontology

OWL documents are XML documents and can contain complex structures. Furthermore, the OWL standard supports many different ways of defining an element. XSLT was designed exactly for the purpose of handling such complex XML structures. Compared to other XML parsers, XSLT is defined in clearly arranged templates to be able to handle these structures and is therefore easy to understand and has a high learning curve. Moreover XSLT is processor-independent, which means that a XSLT template file can be executed on every XSLT compatible processor on every arbitrary operation system. In this implementation the xsltproc tool\(^3\) is used, which is installed on most standard configured Linux systems.

These advantages and the independence of other programming languages and their complex structures or operating system dependencies make the decision for the use of XSLT obvious.

Nevertheless, XSLT is specialized in XML manipulation and therefore not capable of performing further complex processing tasks of the extracted ontology information. These tasks are taken over by the database system using the procedural language of the database system. In figure 5.2 the detailed import process and the included tasks are shown.

\(^2\)http://www.w3.org/TR/xpath
\(^3\)http://xmlsoft.org/XSLT/xsltproc2.html
The import process is subdivided into two parts. The first part, which is performed on the client side, reads the OWL file and converts the ontology information into a database readable SQL format. The next step is to send the SQL file to the database server, where it is processed and stored to the relations.

All subtasks of the first part of the import process are controlled by a shell script which is called by

```
owl2db.sh owl_input.owl sql_output.sql
```
The specified OWL input file is submitted to the xsltproc tool which parses the OWL file and generates SQL output according to the XSLT template file (listed in appendix A.3). This output is saved to the specified output file and is ready to be submitted to the database system. Subsequently, the generated SQL file is submitted to the database for execution by any arbitrary database client.

At the server side a session is created and the received procedure calls are executed within this session. The submitted ontology information is stored in the relations as described in section 5.2. After the insert process, the reasoning process is started. It performs recursive statements (chapter 5 and 6) to derive new facts out of the available ontology information. The obtained information is saved to separated reasoning relations, which contain the original information and the derived facts. The reasoning task is the last task and completes the import process. Thereafter the original ontology information and all derived facts can be directly accessed by SQL statements in a very simple and fast way.

In the following part, the interaction between the client XSLT and the database process is described in detail.

XSLT reads the OWL input file and walks through the OWL structure recursively. At the beginning of each SQL output file (one per ontology) the database procedure init_session is called. The session is needed to store import values during the import process, such as the current ontology ID, XML namespaces or other temporary values. The next step is to call the procedures add_namespace and create_ontology to set the session environment correctly for further ontology information resolving and processing.

Subsequently the processing is started and for each OWL subblock a SQL procedure call with respective arguments is created. Listing 5.2 shows a simple OWL class structure and the resulting SQL procedure calls. The first element <owl:Class rdf:ID="Fish"> results in the SQL code CALL create_term ('Fish', 'class') which inserts a new entry in the relation Term with an unique ID, Term(termID, 'Fish', class). The next subelement <rdfs:subClassOf rdf:about="#Animal"> would be transformed to a call of the procedure create_prop_inst(property, subject, object).
At the point of the XSLT execution, the unique termID of the subject, in the case of listing 5.2 the parent class *Fish*, is not known yet. The termID is created at the point of execution in the database system and can therefore not be used as an argument of the procedure. To cope with this problem, a stack is provided during the session. This stack contains the created unique termIDs and corresponds to the OWL XML hierarchy. This stack provides some functions which allow the calling of the `create_prop_inst` procedure with the correct arguments. Instead of the unique termID, the function `stack_last_id()` is used as the subject argument. At the point of execution, the database replaces the function with the correct termID of the last created term, which at this point of time is on top of the stack.

The `create_term` procedure adds the unique termID to the stack automatically, whereas the `stack_pop` function has to be called manually. XSLT writes the `stack_pop` function to the SQL output file after having processed all subelements of the respective term. In the case of `<owl:Class>` the top element of the stack is popped after having reached `</owl:Class>`. The following listing shows the same scenario using the stack.

```
<owl:Class rdf:ID="Fish">
  <rdfs:subClassOf rdf:about="#Animal">
  </owl:Class>

↓ Meta-Inn Approach Mapping

CALL create_term('Fish','class');
CALL create_prop_inst('subClassOf','Fish','Animal');
CALL stack_pop();
```

Listing 5.2: OWL and corresponding procedure calls

The stack offers the possibility to access already created termIDs of parent-elements. However, some OWL structures require the access to termIDs of child-elements. This problem occurs for example, if the

```
<owl:Class rdf:ID="Fish">
  <rdfs:subClassOf rdf:about="#Animal">
  </owl:Class>

↓ Meta-Inn Approach Mapping

CALL create_term('Fish','class');
CALL create_prop_inst('subClassOf', stack_last_id(), 'Animal');
CALL stack_pop();
```

Listing 5.3: Procedure calls using the stack
object value is not a simple reference, but a new definition, such as shown in listing 5.4. As the inner owl:Class definition is the object of owl:equivalentClass, the definition of the inner class has to be processed before the equivalentClass property can refer to this inner class. This can simply be realized by moving the create_prop_inst call of the inner class after the recursive step into the subelement owl:equivalentClass. If the processing of the subtree of the equivalentClass property is completed, all necessary database entries are available and can be used as an object in the procedure call create_prop_inst(). The problem at this point is the unknown termID of the child-element, because the created termID was already popped from the stack. As the stack only allows access to parent IDs, the stack concept is extended to be able to access already popped IDs (child-elements) as well. For this purpose the function stack_magic_popped_id() returns the last popped termID. In the case of listing 5.4, the created termID of the inner class for the instantiation of the property equivalentClass is needed. This is realized by the procedure call of create_prop_inst as shown in listing 5.4. The function stack_last_id returns the termID of the class AnimalInOcean, whereas stack_magic_popped_id returns the termID of the just created, anonymous inner class.

```xml
<owl:Class rdf:ID="AnimalInOcean">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#Ocean"/>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#livesIn"/>
          </owl:onProperty>
        </owl:Restriction>
        <owl:Class rdf:about="#Animal"/>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

\[\text{Meta-Inn Approach Mapping}\]

Wolfgang Gassler, Eva Zangerle
CHAPTER 5. META-INN APPROACH

Meta-Inn Approach Mapping

CALL create_term('AnimalInOcean','class');
CALL create_term('', 'class'); -- creation inner class
[...]
CALL stack_pop(); -- pops inner class from stack
CALL create_prop_inst(
    'equivalentClass',
    stack_last_id(), -- get termID of class AnimalInOcean
    stack_magic_popped_id() -- get termID of inner class
);

Listing 5.4: Stack utilization

All procedures are available with different signatures to support integer and string arguments. As the procedure overloading support of SQL/PL is very weak, all overloaded procedures or function are named with a signature postfix. E.g. the function fun_sis has three arguments with respective types string (or in database notation VARCHAR), integer and string. If the argument is of type string, the procedures try to resolve the specified name to the corresponding individual/class or rather its respective termID. If the string cannot be resolved, a new literal term entry is created. The object reference of the OWL element <rdfs:subClassOf rdf:about="#Animal"> is resolved to the respective termID of class "Animal", whereas the object of the element <rdfs:subClassOf rdf:about="http://somedomain.com/some.owl #Animal"> results in a new literal term entry, which contains the specified URI.

Property name arguments are manipulated as well. As properties are defined by the namespace and the name itself, the procedures convert the name to a fully qualified URI using the current namespaces defined in the session environment. For example the property rdfs:subClassOf is completed to http://www.w3.org/2000/01/rdf-schema#subClassOf. Fully qualified URIs are unique and can therefore be used in a global scope. Most basic RDF, RDFS and OWL elements (e.g. subClassOf, Class, cardinality, etc), so called "bootstrap terms", are already available in the Term relation and can globally be reused. As the bootstrap terms are globally unique and used in most ontologies, the reuse saves much storage space in the database system. The bootstrap elements are also necessary for the reasoning process, because the reasoning rules are connected to the respective property URIs. Hence only properties with correct, fully qualified URIs are included in the reasoning process.

Another problem of OWL documents is the order of elements. It is possible to refer to other local OWL elements, which are defined after this reference. As the order of the procedure calls is equal to the order of
the definitions in the OWL document, it is possible that references point to other objects, which don’t exist at this point. Therefore the reference resolver creates an entry in the Term relation of type URI, on which the reference points. If the element is defined later, the Term entry of type uri is changed to correct type (e.g. class) and is thereby converted from a simple uri reference to a class instance. As the same entry is used and only the type is changed, already defined references to this entry stay valid and the ontology information keeps consistent.

This chapter was concerned with the Meta-Inn Approach and its implementation. Firstly, the core of the approach, the mapping to the three relations Term, PropertyInst and Restriction, was explained and listed. The next section showed the idea and the implementation behind the import process of ontology data to the database. This part was realized by XSLT and database procedures, which import the OWL ontology to the very fast processable relational database format.

The Meta-Inn Approach is platform-independent as the two main parts, the import process and the database backend, are not restricted to any particular software environment. The only requirements are the existence of an XSLT processor and a common database system supporting recursive SQL and a SQL procedure language. As the approach is based on domain specific programs, it can benefit from all optimization processes of these tools. XSLT can be processed in parallel and executed by high-performance and well-developed XSLT processors. Furthermore, big and highly optimized database systems are responsible for the performance of the backend. Furthermore, it provides all typical database features such as multiuser access, a transactional system, recovery management and optimized disk storage.

The Meta-Inn Approach provides a very simple and standardized interface due to the use of SQL. The reasoning process can benefit from this storage format too as it is also able to reason about very large ontologies on the fly or use concepts such as materialized views or multiblock queries. Even though there are many possibilities for optimizations, the Meta-Inn approach already contains many improvements and advantages over the previously presented approaches. Further implementation details can be found in the appendix.
Chapter 6

Reasoning

The Meta-Inn Approach provides a reasoning engine, which derives new facts from the imported ontology information. The reasoning support is implemented in SQL and offers some basic reasoning features. The reasoning process consists of the following five inference rules (presented in Prolog coding style), which are applied to the imported data.

Symmetric Properties

\[
\text{PropertyInst}(P, O, S) :\neg
\text{PropertyInst(type, P, SymmetricProperty)},
\text{PropertyInst}(P, S, O).
\]

The above rule searches for instantiations of symmetric properties and creates a new instance with swapped subject and object. A typical symmetric property is the "complementOf" property, which is used in the following example.

\[
<\text{owl:Class} \text{ rdf:ID="Land">}
<\text{owl:complementOf} \text{ rdf:about="#Water"/>}
</\text{owl:Class}>
\]

\text{↓ Meta-Inn Approach Mapping}

\text{PropertyInst(complementOf, Land, Water)}

\text{↓ Meta-Inn Approach Reasoning}

\text{PropertyInst(complementOf, Land, Water)}
\text{PropertyInst(complementOf, Water, Land)}
CHAPTER 6. REASONING

Transitive Properties

\[ \text{PropertyInst}(P, S, O) :\]
\[ \quad \text{PropertyInst}(\text{type}, P, \text{TransitiveProperty}), \]
\[ \quad \text{PropertyInst}(P, S, X), \]
\[ \quad \text{PropertyInst}(P, X, O). \]

Transitive reasoning is defined by the above rule, which detects all transitive paths between transitive properties. Hence if property \( P \) holds for subject \( S \) with object \( X \) and \( X \) has the same property \( P \) with object \( O \), the fact that also \( S \) has the property \( P \) with object \( O \) can be derived. One of the most fundamental transitive properties is the "subClassOf" property, which is shown in the following example.

\[
<\text{owl:Class rdf:ID}="\text{Fish}">\\
<\text{rdfs:subClassOf rdf:about}="#\text{Animal}"/>\\
</\text{owl:Class}>
\]
\[
<\text{owl:Class rdf:ID}="\text{Clownfish}">\\
<\text{rdfs:subClassOf rdf:about}="#\text{Fish}"/>\\
</\text{owl:Class}>
\]

\( \downarrow \) Meta-Inn Approach Mapping

\[
\text{PropertyInst}(\text{subClassOf}, \text{Fish}, \text{Animal})\\
\text{PropertyInst}(\text{subClassOf}, \text{Clownfish}, \text{Fish})
\]

\( \downarrow \) Meta-Inn Approach Reasoning

\[
\text{PropertyInst}(\text{subClassOf}, \text{Fish}, \text{Animal})\\
\text{PropertyInst}(\text{subClassOf}, \text{Clownfish}, \text{Fish})\\
\text{PropertyInst}(\text{subClassOf}, \text{Clownfish}, \text{Animal})
\]

Inverse Properties

\[ \text{PropertyInst}(I, O, S) :\]
\[ \quad \text{PropertyInst}(\text{inverseOf}, P, I), \]
\[ \quad \text{PropertyInst}(P, S, O). \]

If there exists an inverse property \( I \) of property \( P \), for each property instantiation of property \( P \), a new entry of property \( I \) is created. The new entry consists of swapped subject and object of the original property \( P \), as shown in the following example.
CHAPTER 6. REASONING

Property Inst (inverseOf, livesIn, isHabitatOf)
Property Inst (livesIn, Nemo, Pacific)

Meta-Inn Approach Reasoning
Property Inst (inverseOf, livesIn, isHabitatOf)
Property Inst (livesIn, Nemo, Pacific)
Property Inst (isHabitatOf, Pacific, Nemo)

Property Inheritance
Property Inst (P, S, O) :-
   Property Inst (subClassOf, S, X),
   Property Inst (P, X, O).

Property Inst (P, S, O) :-
   Property Inst (equivalentClass, S, X),
   Property Inst (P, X, O).

P...unionOf, oneOf, intersectionOf

The three properties unionOf, oneOf and intersectionOf are inherited by subclasses, because they define the class structure and behaviour. Therefore the subclasses have to fit the schema of the superclass. The same holds for equivalent classes. The following example shows how unionOf definitions are derived to subclasses.

...
CHAPTER 6. REASONING

Meta-Inn Approach Mapping

PropertyInst (unionOf, AnimalFood, FoodInWater)
PropertyInst (unionOf, AnimalFood, FoodOnLand)
PropertyInst (subClassOf, FishFood, AnimalFood)

Meta-Inn Approach Reasoning

PropertyInst (unionOf, AnimalFood, FoodInWater)
PropertyInst (unionOf, AnimalFood, FoodOnLand)
PropertyInst (subClassOf, FishFood, AnimalFood)
PropertyInst (unionOf, FishFood, FoodInWater)
PropertyInst (unionOf, FishFood, FoodOnLand)

Restriction Inheritance

Restriction (P, S, R) :-
    PropertyInst (subClassOf, S, X),
    Restriction (P, X, R).

Restriction (P, S, R) :-
    PropertyInst (equivalentClass, S, X),
    Restriction (P, X, R).

A restriction R on property P in class X is inherited to all subclasses S. The same holds for equivalent classes. The following example shows such a subclass restriction inheritance.

<owl:Class rdf:ID="Dolphin">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:hasValue>true</owl:hasValue>
            <owl:onProperty>
                <owl:FunctionalProperty rdf:ID="canSwim"/>
            </owl:onProperty>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

Meta-Inn Approach Mapping

PropertyInst (subClassOf, Dolphin, Restriction123)
Restriction (canSwim, Restriction123, true, hasValue)

Meta-Inn Approach Reasoning

PropertyInst (subClassOf, Dolphin, Restriction123)
Restriction (canSwim, Restriction123, true, hasValue)
Restriction (canSwim, Dolphin, true, hasValue)

Listing 6.1 shows the output of the class description of the individual Pacific of the example ontology listed in appendix A.6. The first request
is performed on the unreasoned data, which were imported from the OWL file. It is only known, that the individual Pacific is of type Ocean, as it is connected to it by a subclass relation. If the same description command is executed on the reasoned data, the result set contains much more information. The derived data consists of all subclass relations and property instances of type isHabitatOf. As there are no isHabitatOf instances in the original ontology, the instances are derived from the inverse property livesIn, which is specified for all animal individuals.

```
-- sets the specified ontology to the
-- default ontology of the current session
CALL set_ontology('AnimalExampleOntology');

SELECT * FROM TABLE(describe_class('Pacific'));

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PROPERTY</th>
<th>PROPERTYVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacific</td>
<td>http://... rdf-schema#subClassOf</td>
<td>Ocean</td>
</tr>
</tbody>
</table>

SELECT * FROM TABLE(describe_reasoned_class('Pacific'));

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PROPERTY</th>
<th>PROPERTYVALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacific</td>
<td>http://... rdf-schema#subClassOf</td>
<td>Habitat</td>
</tr>
<tr>
<td>Pacific</td>
<td>http://... rdf-schema#subClassOf</td>
<td>Water</td>
</tr>
<tr>
<td>Pacific</td>
<td>http://... rdf-schema#subClassOf</td>
<td>Ocean</td>
</tr>
<tr>
<td>Pacific</td>
<td>isHabitatOf</td>
<td>TheWhiteShark</td>
</tr>
<tr>
<td>Pacific</td>
<td>isHabitatOf</td>
<td>Flipper</td>
</tr>
<tr>
<td>Pacific</td>
<td>isHabitatOf</td>
<td>Marlin</td>
</tr>
<tr>
<td>Pacific</td>
<td>isHabitatOf</td>
<td>Nemo</td>
</tr>
</tbody>
</table>
```

Listing 6.1: Output of Pacific before and after the reasoning process

### 6.1 Implementation

The reasoning part is completely implemented in SQL and uses recursive SQL statements to derive new facts. The resulting reasoned data can be saved to a new relation or can be made accessible by a view, which recomputes the new fact with each request. The following part compares these two approaches.

- **Views** are very easy to administrate, because the view always operates on the current data and does not need recalculation schedules to cope with updates or inserts on the original relation. An-
other advantage is the storage size of the database, which is not influenced by the view. Reasoned data of even small ontologies can grow exponentially and need a large amount of disk space. Views are calculated on the fly with each request and do not need any additional disk space. Although this advantage can lead to a big problem in the case of the reasoning process, because the recursive reasoning statements are very complex and can bring down the performance.

- **Separated relations** As already described, a separated relation to store reasoned data needs a mechanism to recalculate the inferred data on updates or inserts of the original relations. The recalculation can be realized by a new full restart of the reasoning process or a smart adaption of the reasoned data according to the changes on the original relations. Another problem is the size of the new relation, which can grow exponentially. Even small ontologies can result in an additional usage of a very big amount of disk space. In return the access-performance is very high, because no further calculations are associated with a request and the reasoned data can be accessed directly.

![E/R diagram of the reasoning relations](image)

Figure 6.1: E/R diagram of the reasoning relations

The Meta-Inn Approach reasoning engine is realized by using two new relations (see figure 6.1) to store the derived data. As the reasoning
CHAPTER 6. REASONING

statements are very complex, the implementation with separated relations is more convenient. The reasoning process is started after each completed import process and loops over all inference rules until no additional facts can be discovered. The loop is necessary to derive facts, which are based on other inferred data. For example, new transitive relations can be encountered after applying the symmetric rule or vice versa.

The following two procedures control the reasoning process. The procedure \texttt{do\_reasoning} uses already derived data and tries to discover new facts. It should be called after inserting new ontology information. The import process calls this procedure after the completed import of a new ontology. If any changes are made or facts are deleted, the procedure \texttt{refresh\_reasoning} has to be called to rebuild the reasoned facts. As ontologies contain many complex dependencies, it is very difficult to compute all entries affected by a simple update or delete query. Therefore the procedure \texttt{refresh\_reasoning} deletes all inferred facts and rebuilds the reasoned relations by calling the \texttt{do\_reasoning} procedure. At the beginning of each reasoning process, the available facts are copied from the original relations to the respective reasoned relations to provide a basis for the reasoning computations.

The relations \texttt{PropertyInstReasoned} and \texttt{RestrictionReasoned} contain all reasoned facts and almost have the same schema as the relations \texttt{PropertyInst} and \texttt{Restriction}. As reasoned data are not updated and do not have to be accessible by an unique ID, the primary unique key column of both reasoned relations is removed. For performance reason the foreign key checks are also deactivated. To allow fast access, indices are created on the property and the subject columns, on which most queries are based.

Listing 6.2 shows the recursive SQL query to derive new facts using transitive relations. The first select statement (line 4) fetches all available facts from the \texttt{PropertyInstReasoned} relation. The recursive step (line 9), which is connected with an \texttt{UNION ALL}, joins the current working relation \texttt{reason} with \texttt{PropertyInstReasoned} and searches for property pairs, which are connected by subject and object. Only properties, which are \texttt{subClassOf} or \texttt{type} of \texttt{TransitiveProperty} are taken into account by the subquery in line 17. If such a pair is found, a new property instance with subject of the first entry and object from the second entry is created.

The problem of such recursive statements are circle paths, which lead to an infinite loop and the non-determination of the SQL query. To avoid this problem, an additional iteration counter is added. The counter named \texttt{iterations} is incremented with each recursive step. The condi-
CHAPTER 6. REASONING

... iterations+1 < 10 breaks the recursive execution after ten iterations. Therefore only paths of the maximum length of ten are discovered. This setting should suffice for most ontologies, otherwise the value can be increased. The type of basic OWL properties (e.g. subClassOf is transitive) is defined in a bootstrap set of terms and relations, which is available in a global scope. These bootstrap terms can be used by every ontology and are therefore used to select the specific properties by the reasoning process (e.g. line 20-31 in listing 6.2).

```
WITH reason (propid, propinstsub, propinstobj, iterations) AS
  (SELECT p.propid, p.propinstsub, p.propinstobj, 0
   FROM propertyinstreasoned p
   UNION ALL
   SELECT r1.propid, r1.propinstsub, r2.propinstobj, iterations+1
   FROM reason r1, propertyinstreasoned r2
   WHERE r1.propinstobj = r2.propinstsub
   AND iterations+1 < 10
   AND r1.propid = r2.propid
   AND r1.propid IN
   (-- get all transitive properties
    SELECT t.propinstsub
    FROM propertyinstreasoned as t
    WHERE
    (t.propid = get_term_id('http://.../22-rdf-syntax-ns#type',
      'property')
    OR t.propid = get_term_id('http://.../rdf-schema#subClassOf',
      'property')
    AND t.propInstObj = get_term_id('http://.../owl#TransitiveProperty',
      'property')
  )
  AND r1.propInstSub <> r2.propinstobj -- to avoid self relation circles
  )
SELECT * FROM reason
```

Listing 6.2: Transitive rule implemented in SQL

Wolfgang Gassler
This chapter was concerned with the Meta-Inn Approach reasoning engine and its implementation. The reasoning process consists of five rules, which are able to derive new facts. The rules can discover relations by transitive, inverse or symmetric paths and inherit properties and restrictions from superclasses. The inferred information is stored in two separated relations and can be accessed directly.

The reasoning engine is realized in a simple way, as the rules are completely defined using SQL. Furthermore, this strategy can benefit from the complex SQL optimization process and the highly developed data handling of the database management system. As SQL is very powerful and provides a standardized interface, there are many possibilities for future extensions.
Chapter 7

Mobile Computing and Mobile Databases

Especially during the last few years, many applications were adapted for the use in mobile environments. This can be lead back to the fast growth of mobile technologies and mobile devices. People more and more felt the need to be able to work or even play everytime and anywhere. Due to this mobile boom many new fields of application evolved. The area of mobile gaming and location based services are two examples of these fields and represent a completely new sector enabled by the capabilities of mobile computing.

The development of mobile applications has also brought up many challenges, which lead to new approaches and programming paradigms. The two fundamental shortcomings are the restricted connectivity and the size of the device. The size of the device puts constraints on the display size, processor speed and the size of the memory.

Former mobile applications were based on a client-server architecture, which requires a constant connection between client and server. Despite the fact that the wireless network coverage is increasing heavily, a constant connection cannot be realized. There are too many influencing factors, such as dead spots, tunnels or other interfering obstacles. In some places, e.g. on an airplane, an online connection is still not possible at all. Especially for applications which need a connection to the central server in order to work correctly, the “not–always–on” issue is a crucial factor. To enable mobile applications that are not restricted by the need of a constant connection to the server, the concepts of replication and synchronization are playing an important role.

In the following sections, such concepts and new approaches are explained in detail.
CHAPTER 7. MOBILE COMPUTING AND MOBILE DATABASES

7.1 Mobile Databases

As databases have become a standard solution for data storage, it is an obvious choice to adapt databases and their concepts for mobile environments to provide a well-known and common interface to a secure storage management system. Traditional database systems are very resource-intensive and are mostly run on big server systems, which provide a huge amount of storage space, main memory and processor capacity. In contrary to traditional environments, mobile devices have very limited resources. Therefore many adaptations are necessary to transform a traditional database system into a mobile database system. The first restriction is put on the size of the executable application itself. This size is called "footprint" and can for example require between a few hundred Kbyte for PDAs and a few Kbytes for Smartcards, which are called "Pico Databases". Furthermore, the main memory and processor performance are limited and this fact influences the possible size of the intermediate results of a query and its processing and optimization strategy. The storage strategies and index concepts also have to be suitable for the mobile environment. Beside the local modifications as described above, the communication with the central server includes many new concepts concerning the synchronization and replication process.

The synchronization process aims at preserving a consistent database state at all distributed instances. There are many different synchronization algorithms to reach this consistency, which can be classified into reintegration and reconciliation. The reintegration process propagates any changes on the slave side (sink) to the master side (source). The reconciliation process merges the changes propagated by the clients and broadcasts all master side changes to the sinks. If reintegration and reconciliation are executed consecutively, the resulting process is called "bidirectional synchronization".

The synchronization algorithms, which are used in mobile environments, are mostly based on traditional synchronization algorithms, such as [Dsd06, Hœp01, Dat95, EN]. Most methods rely on optimistic locking strategies, which do not set up locks and check for possible conflicts later. Only in case of a conflict different conflict resolution strategies are applied. On the other side pessimistic locking creates locks and prevents any conflicts. As locks could remain active due to the loss of the connection for a very long time, mostly optimistic locking methods are used when dealing with mobile databases. In the following section the most important concepts of synchronization and replication for mobile databases are explained.
7.1.1 Virtual-Primary-Copy

The Virtual-Primary-Copy technique is a pessimistic strategy and extends the traditional Primary-Copy strategy. Every client creates a copy of the master and only alters the local copy if a lock can be established. Therefore it is not possible to get any inconsistencies. As this method requires a connection to the master, problems could evolve if the master is based on a mobile device. If the master is disconnected, the clients are not able to create locks. To solve this problem, one virtual primary copy of the master is placed on a wired device, which can temporarily replace the master. After the original master has set up a connection again, the two versions have to be merged.

7.1.2 Snapshot Strategies

Snapshots are locally materialized views of parts of the remote database. The views can be available in different granularities, which can consist of a whole relation or just a few tuples. The manipulations are applied to the local snapshot and are propagated to the master side in arbitrary time intervals. Possible conflicts are solved by adequate conflict resolution routines. The communication can be done connection-oriented (synchronous) or message-oriented (asynchronous). The Snapshot communication is often based on the Publish and Subscribe model. In this model the master publishes different views (articles), which can be subscribed by the slaves. These articles can be grouped to logical entities, which are named publications.

Snapshots can be divided into simple and complex Snapshots. Simple snapshots only contain data originating from one relation and therefore a bijective mapping between the tuples on the master and slave side is available. Complex snapshots can result from joins, aggregations or other complex operators on various relations or other snapshots. As there is no way to find a bijective mapping between the snapshot tuples and the original tuples on the master side, reconciliation is not possible. Therefore complex snapshots are only used in read-only-mode on mobile clients, whereas simple snapshots can be reconciled with appropriate strategies.

7.1.3 Replication Proxy Servers

The concept of Replication Proxy Servers is a three tier architecture that defines a middleware layer between the mobile clients and one or more database servers. It provides a standardized interface to all clients and controls the complete synchronization and replication process. Therefore it is possible to develop complex and custom synchronization and replication methods, which are independent of the underlying database.
systems. A copy of the original database is maintained on each proxy server and is used to handle all queries, replication requests and synchronization processes with the clients. This proceeding requires additional synchronization between the proxy database and the original database. The proxy server can either be located on the same host as the database management system or run on one or more dedicated machines. This splitting has the advantage of being able to balance the load and increase the scalability, especially in the case of many involved clients and synchronization algorithms with high computational complexity.

7.1.4 Synchronization Markup Language (SyncML)

SyncML is a standardized protocol to handle synchronization and replication and is developed by the Open Mobile Alliance \(^1\), which consists of most of the leading mobile device companies. SyncML uses XML and consists of two subprotocols, SyncML Sync Protocol and SyncML Representation Protocol. The SyncML Sync Protocol is responsible for the connection between two parties, which comprises e.g. connection establishment, authentication or error handling. The other protocol, SyncML Representation Protocol, defines the transport of data and all processes related with the synchronization and replication. SyncML only defines the protocol and not the format of the data, which has to be synchronized. Therefore it is possible to build up on already existing formats, which simplifies the development process. Conflict resolution is also handled very flexible, because it can be defined by the user and is not limited by the protocol specification. This part of the server program is called Sync Engine and is responsible for the resolution of conflicts and the information of clients about a resolution or an unresolvable conflict. SyncML offers the following status codes, which are sent to the clients by the Sync Engine.

- **Merge:** The conflicting client and server entries are merged into one entry. The merged data is then sent back to the client.

- **Duplicates:** The entries standing in conflict are duplicated, which results in two entries in the server database. The server propagates the new entry to the client side resulting in two identical entries on both sides.

- **Winner:** The Sync Engine decides which side is more important and therefore "wins" the conflict. If the server is the winner, the client has to do an "undo" on the changes.

\(^1\)http://www.openmobilealliance.org/
In this chapter mobile environments and mobile databases were introduced. In the last part, various concepts for the adaption of common databases to be able to cope with the special conditions of mobile environments were explained.
Chapter 8

Ontologies in Mobile Environments

Current projects dealing with mobile clients are based on a client-server architecture and move the computational effort to the server, where resources are hardly limited. Ontology based query rewriting [WGV06] is a typical example for such an approach. This rewriting approach is for example used for the Olympic Games 2008 in Beijing, China. The framework, which is called FLAME2008, provides personalized, situation-aware services in push and pull mode to deliver contents to the mobile clients. Beside already known techniques, such as location-based and context-aware services, situation-aware processing was integrated. Situation-aware processing means that specific contexts are translated into logical situations, which can be used to refine the answer to the mobile request. All these customizations are realized by using ontologies. Currently there are hardly any applications which process ontology information on mobile devices. Due to the fact that reasoning is very complex and mobile devices have limited capacities, the workload was moved to the server. During the last years a very fast development of mobile CPUs could be observed. At the moment modern PDAs are built on RISC processors with about 600 MHz and provide up to 128 MB main memory. Secondary memory, such as flash memory cards and mini hard disk drives, just reached a capacity of 16 GB and can therefore store big amounts of data.

The steadily increasing performance of mobile devices offers the possibility of distributing parts of the server workload to the clients. As the number of clients rises constantly, the exploitation of the involved resources (server and clients) is indispensable. Consider the example of a server, which handles 1000 clients and a possible outsourcing of 20% of the computational effort to the clients. After this load distribution, the server can handle 200 additional clients.
Especially in the case of reasoning about ontologies, which is very resource-
greedy, a workload distribution can result in a performance boost even
when only a small part can be outsourced. Additionally the distribution
relieves the wireless connection, because there is only one connection
establishment to receive the basic data set and all further operations are
done locally on the client.

In most applications the process of creating and manipulating ontolo-
gies is a manual task, which can only be done efficiently on a device
with sufficient screen size and therefore is not appropriate for the use on
current mobile devices. The processing of ontologies, e.g. the rewriting
of requests [NF03] can be handled on mobile devices because the pro-
cessing does not require any interaction with the user or visualization of
ontologies. Although the creation and manipulation process on mobile
devices is only possible, if it is handled automatically. An application
could deduce new knowledge out of user input and behaviour and insert
it to the underlying ontology, where the user does not even have to no-
tice the existence of the ontology. In this area many applications, which
require transformation algorithms from user input to ontology knowl-
dge [WGV06] are conceivable. As already shown, the application of
ontologies in mobile environments is reasonable and therefore requires
suitable concepts to process ontologies in a fast and reliable way. The
next section presents a solution, which fulfils these requirements.

8.1 Ontologies in mobile Environments using
mobile databases

As shown in chapter 4, databases can be used to process and store on-
tologies. Chapter 7 shows that mobile databases are very well suitable
to manage data on mobile devices and can replace the old concept of
client-server architecture in some areas. According to the previous sec-
tion, ontology processing on mobile devices has become more and more
important and needs an underlying reliable and stable high-performance
system. It would be an obvious choice to use mobile databases, which
would meet all these requirements. As the Meta-Inn Approach is based
on databases, it provides a basis to build up an ontology processing
system based on mobile databases.

Basically there are two ways of implementing the reasoning part of the
Meta-Inn Approach in mobile environments. The first approach is to
reason about ontologies on the server and store all derived facts in views
or separated relations. These views or parts of the ontology are published
in publications (see also section 7.1.2), which can be downloaded by the
clients. The clients operate on these snapshots and can therefore access
and query all data locally as complex computations have already been calculated on the server. Even so the server and the wireless connection get relieved as the data only has to be submitted once and all further querying can be done locally on the client.

The second approach is to execute the complete reasoning process on the client. Therefore it is necessary to transfer the ontology or a part of it to the client – depending on the bandwidth and the storage capacity. The client is able to reason locally, without the need of a connection to the server and therefore relieves the server. In contrary to the first approach, client based reasoning requires a certain amount of disk space and processor power. On the other hand the load on the server and the wireless connection decreases, because only the basic ontology definitions get transmitted and there is no need for complex reasoning on the server. Table 8.1 compares the two approaches.

<table>
<thead>
<tr>
<th></th>
<th>server-sided reasoning</th>
<th>client-sided reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>server load</td>
<td>higher load</td>
<td>lower load</td>
</tr>
<tr>
<td>client load</td>
<td>lower load</td>
<td>higher load</td>
</tr>
<tr>
<td>inference duration</td>
<td>low (high computation power)</td>
<td>high (limited capacities)</td>
</tr>
<tr>
<td>reasoning</td>
<td>full</td>
<td>partial (limited space)</td>
</tr>
<tr>
<td>connection establish- ments</td>
<td>more</td>
<td>less</td>
</tr>
<tr>
<td>connection load</td>
<td>high (basic + derived data)</td>
<td>low (basic data)</td>
</tr>
</tbody>
</table>

Table 8.1: Comparison of Client- and Server-sided reasoning

As the two approaches both have advantages and disadvantages, there are some possible combinations and improvements of the two approaches above, which try to exploit the advantages of the respective approaches.
in an enhanced way. All limitations of server-sided reasoning can be lead back to the ontology size, as the derived data can exceed the available disk storage. To avoid this problem, there are many improvements, which were developed in the area of data warehouses as these have to deal with huge amounts of data as well. Also the fact that mostly read-only-transactions are executed on the underlying data leads to the use of data warehouse specific algorithms. For example the usage of materialized views [BG04] and multiblock queries [BG04] can solve the limitation mentioned above. Frequently requested derived facts are stored as materialized views, the remaining part is computed dynamically as soon as the data is inquired.

There are also some improvements regarding the distribution of the computation. In the first approach the whole computation is done on the server side, whereas in the second approach the client itself computes the derivations. It is also possible to distribute the workload to the server and the client, which means that the client only does the simple computations and the complex work is done by the server. The server is able to handle these complex queries because it can directly use the whole ontology knowledge and has more resources available. Typical complex queries are aggregations, where a big amount of data is used to compute a relatively small result, for example a query counting the number of all existing Fish in the example ontology. If the mobile approach would be used in this example, all Fish instances would have to be transferred to the mobile client to enable the client to compute one single value.

Another problem is the granularity of the data of a publication, which influences the amount of data which has to be transmitted. Let’s consider a publication containing all fish and a client, which wants to show a list of all clownfish. Therefore the client would have to pass through the whole list and check for each entry, whether it is a clownfish or not. To fulfill this task, the client has to retrieve the whole publication. If the client would only retrieve a part of the publication, it could not make sure that all clownfish were encountered. To avoid this connection overload by downloading publications containing unnecessary data, a dynamic adaption of the publications requested by the client is needed. For example, PointBase Micro [Mic07] implements this approach using filter rules, which are defined by the clients. The client sends a set of filter rules to the server, which executes these rules on the publications and sends the filtered data back to the client.

A common problem of the two approaches is to determine the data that has to be transferred to the client. If the reasoning is carried out on the client, it has to request new data sets with each iteration of
the reasoning process. To reason for example about a simple subclass-hierarchy of a class, the client first has to obtain all the information about the class itself. This step is repeated for every encountered parent class and therefore new information has to be requested during every iteration. Based on the newly gathered information, the client has to repeat the reasoning process as all the information has to be taken into account until no more new facts can be discovered. This leads to many requests and a high transfer volume, which results in a bad performance and no advantages of client-sided reasoning. Figure 8.1 shows the sets of information that are requested by the client during the first three steps of the reasoning process. The starting point is a class in the center, on which all possible facts should be discovered. Each recursive step to discover more facts consists of at least one request, but this number could increase if the structure of the ontology is very complex.

If the reasoning process is executed by the server, a similar problem is encountered, as the server is not able to precisely identify the data set that is transferred to the client. If a client requests information about a certain class, the server has to take into consideration all information related to the specified class. As this would lead to huge data sets which
could overstress the client’s abilities, the server has to choose a subset of the full result set that is transmitted to the client. To decide such a subset, the server has to know which information are relevant for the client.

Figure 8.2: Possible choices of result sets, transmitted to the client

Figure 8.2 illustrates a single class (center) and all classes or properties, which are connected or related to the instance. Let’s consider a client request to retrieve information about the marked example class in the center. The maximum number of elements that the client can deal with is five. In the figure some of the possible ways of choosing five elements are marked. For example, the path of a subclass hierarchy request is very different (depth-first search) from requesting all properties of a certain class (breadth-first search). This problem could be solved by inventing a communication protocol, which provides the possibility for the client to specify the way of how the server chooses the relevant elements of the ontology.

Basically the reasoning process can completely be executed on the client or on the server side, however also a mixture of both approaches is
possible. The actual choice of the location of the reasoning process and the distribution of the work depends on the application and the application area. Even though all three approaches require an extended protocol to describe ontology subsets in a proper semantic way, that only expedient data are transmitted and the connection is relieved.

8.2 Merging

Especially in the mobile sector, the server has to deal with various copies on multiple clients, which irregularly transmit the changed data. Therefore it is crucial that there exists a method to reconcile the received changes from the mobile clients.

Ontology mediation currently is a very popular research area [dBEF+06]. It is concerned with the detection of differences between ontologies and ways to solve the problems resulting from this heterogeneity. The term ontology mediation consists of three application areas:

- **Ontology Alignment** is concerned with the detection of similarities or correspondences between different ontologies. It is the basis for ontology mapping and merging, which are the two main approaches for the merging of ontologies.

- **Ontology Mapping** results in a set of mapping rules between corresponding elements of two ontologies. The first step is the discovery of corresponding elements (Ontology Alignment) between two input ontologies. For all the discovered elements, rules that map elements of the first ontology to the second ontology and vice-versa are defined. The set of all resulting rules is the output of the ontology mapping process. The mapping process is illustrated in figure 8.3.

"Figure 8.3: Ontology Mapping Process"
• **Ontology Merging** creates a new ontology based on the input ontologies. There are two ways to merge ontologies:

  – The ontology resulting from the merge of two or more input ontologies replaces the original ontologies.
  
  – The original ontologies are not replaced but the result of the merging process is a bridge ontology, which imports the input ontologies and defines bridge axioms. These axioms define the correspondences between the input ontologies.

As databases are able to handle a huge amount of clients, the ontology mapping approach, which has to store all different user changes and resulting versions, is not feasible. Therefore only the approach “Merging with Replacement” can be used to develop a merging algorithm for ontology processing systems based on databases.

To reconcile changed tuples, the server of mobile database systems needs an algorithm to perform the reconciliation process correctly and preserve the data consistency. To define such an algorithm, the described approaches can only be incorporated in a very limited way, as they are designed to detect differences and similarities of ontologies of different structures, names and hierarchies. The difference detection is not needed by the Meta-Inn Approach, as all differences are known to the database system. Differences consist of changed, deleted or inserted entries, which are all announced to the reconciliation process by the client’s mobile database automatically.

Therefore only the determination of similarities or changes on the same parts of ontologies would be very helpful to implement merging rules. The problem is that all currently available merging tools are semi-automatic and need further assistance by the user. This means that proper rules for an automatic merging can only be defined with additional semantic knowledge about the saved and changed data. Although some basic conflicts can be solved automatically as described in the section below.

Let’s consider two mobile clients, simultaneous changes of class C and the creation of two new subclasses of C named D and E, as shown in figure 8.4.
The problem with this simultaneous change of C is that the subclasses D and E are derived from different versions of C named C' and C''. Therefore the changes of C' and C'' can not be simply applied to the original version C, as shown in the conjunctive merging approach in figure 8.5. Conjunctive merging would provide the best merging result, as the subclasses D and E can directly be attached to the changed version of C. Unfortunately this merging approach is nearly impossible. There are too many obscurities about the semantic meaning and the conjunctive merging can also results in incompatibilities with other subclass instances or relations. Furthermore, the semantic consistency could be destroyed as the server is not able to know which semantic changes the client’s user wants to achieve.

The second merging approach in figure 8.5 retains both client views of the ontology and merges them into one structure. Hence the changed C-versions C' and C'' become subclasses of C, to which the classes D and E are attached. But also the disjunctive merging approach can lead to some conflicts. For example the removal of a property of class C would lead to the fact that the changed class is no longer a proper subclass of C. Therefore the changed class C'' can not be attached to the original class C by a subclass relation. This problem can be solved by the third approach, named Generalisation. If the changes to the class C lead to a more generalised class C''', the new class C''' has to be placed between the original class C and its superclass A (third approach in figure 8.5).

All presented combination approaches are based on syntactic analyses and do not incorporate semantic informations. Although the syntactical consistency is warranted, the semantic correctness can be destroyed by such a combination process.

Wolfgang Gassler, Eva Zangerle
In this section the area of ontology merging was presented. It is a huge research area, which consists of the three application areas Ontology Alignment, Mapping and Merging. It was shown that the ontology processing approaches based on databases are more convenient, as differences and changes are known to the reconciliation process automatically. The definition of proper merging rules without any semantic knowledge is currently impossible and one of the biggest problem concerning merging technologies. Therefore also the reconciliation process of mobile databases need the interaction of the user to resolve conflicts in a proper way.
Chapter 9

Conclusion and Outlook

In many areas of research, ontologies have been getting more and more important. Therefore it is indispensable that even big ontologies can be stored, processed and reasoned about efficiently.

Three basic approaches for the storage of ontologies in databases were presented. Based on the previous approaches, the completely new developed "Meta-Inn Approach" tries to fix the shortcomings of the three approaches. The Meta-Inn Approach is based on well-developed systems as only XSLT and SQL are used. Therefore the implementation can benefit from the underlying systems and its optimizations.

The reasoning engine of the Meta-Inn Approach consists of some basic reasoning rules, which are defined using SQL. As SQL is a powerful and standardized language, new reasoning features can easily be developed. For instance, consistency and satisfiability checking rules or other, more complex rules are possible. Furthermore, a mapping from rules defined in SWRL (Semantic Web Rule Language) to SQL to extend the reasoning engine is conceivable.

The current version provides a simple SQL interface to access the imported and reasoned ontology information. This interface could also be extended to support the widely-used SPARQL query language by defining an appropriate mapping to SQL.

Databases have already found their way to the mobile market, which has grown very fast within the last few years. The main differences between mobile and common database systems are concerned with the replication and synchronisation processes to synchronise the main server and all mobile clients.

As mobile databases provide the same well-known interface as common database systems, to check whether porting the Meta-Inn Approach to the mobile environment is possible, would be an obvious consequence. Subsequently, the Meta-Inn Approach would be able to replace the
widely used client-server paradigm. As shown in the previous chapters, the workload can be transferred from the server to the client to exploit the full potential of the involved resources on the server- and the client-side. The distribution of tasks between the client and the server depends on the ontology, the query type (e.g. aggregations) and the field of application and needs more practical tests to be able to make reasonable propositions.

For a reasonable communication between client and server, a suitable and enhanced communication protocol to define semantic correlations has to be developed. It would define a query in a more detailed semantic way and would lead to an optimized and adjusted communication consisting of expedient data only.

Another crucial topic, which comes into play when dealing with distributed systems such as mobile databases, is the merging process, which centrally reconciles the changes made by the clients. An automated merging process would need rules, which define how changes of ontology knowledge are merged. As ontologies contain semantic knowledge, syntactical rules are not sufficient. A reconciliation is only possible if further semantic knowledge is provided or the process is performed manually.

Therefore the mobile implementation would need more enhancements to cope with the very different mobile environment, although the Meta-Inn Approach and its related concepts showed that database management systems can efficiently be used to store, process and reason about ontologies.
Appendix

A.1 Used Software and Tools

The following software and tools were used during the course of this thesis:

- DB2 Express-C v9.1.0.0
- Aqua Data Studio 4.7.2 - http://www.aquafold.com
- XsltProc (compiled against libxml 20628, libxslt 10121, libexslt 813 and libxml 20628) http://xmlsoft.org/XSLT/xsltproc2.html
- Dia 0.95-1 - a program for drawing structured diagrams http://www.gnome.org/projects/dia/
- WonderWeb OWL Ontology Validator, developed by the University of Manchester and Karlsruhe, 2003 http://www.mygrid.org.uk/OWL/Validator
- Subversion 1.4.3 - http://subversion.tigris.org
- Wikka Wakka Wiki 1.1.6.2 - http://www.wikawiki.org

A.2 Database Schema

The following listing shows the database schema used for the implementation of the Meta-Inn Approach. Further details about the database implementation can be found in section 5.2.
APPENDIX A.2. DATABASE SCHEMA

CREATE TABLE STACK (
    termID INTEGER NOT NULL,
    termTypeName CHAR(10) NOT NULL,
    orderID SMALLINT NOT NULL,
    active SMALLINT NOT NULL DEFAULT 1,
    PRIMARY KEY (orderID)
);

CREATE TABLE TMP (
    k VARCHAR(20),
    v INTEGER
);

CREATE TABLE TMP_CHAR (
    k VARCHAR(100),
    v VARCHAR(1024)
);

CREATE TABLE term (
    termID INTEGER NOT NULL,
    termOntID INTEGER NOT NULL,
    termName VARCHAR(1024) NOT NULL,
    termTypeID INTEGER NOT NULL, --class, prop, value
    PRIMARY KEY (termID),
    FOREIGN KEY (termTypeID) REFERENCES termType (termTypeID) ON DELETE CASCADE,
    FOREIGN KEY (termOntID) REFERENCES term (termID) ON DELETE CASCADE
);

CREATE TABLE termType (
    termTypeID INTEGER NOT NULL,
    termTypeName CHAR(10),
    PRIMARY KEY (termTypeID)
);

CREATE TABLE PropertyInst (
    propInstID INTEGER NOT NULL,
    propID INTEGER NOT NULL,
    propInstSub INTEGER NOT NULL,
    propInstObj INTEGER NOT NULL,
    PRIMARY KEY (propInstID),
    FOREIGN KEY (propID) REFERENCES term (termID) ON DELETE CASCADE,
    FOREIGN KEY (propInstSub) REFERENCES term (termID) ON DELETE CASCADE,
    FOREIGN KEY (propInstObj) REFERENCES term (termID) ON DELETE CASCADE,
);
CREATE INDEX PropertyInstPropID_IDX
ON PropertyInst(propID);

CREATE TABLE PropertyInstReasoned (propID INTEGER NOT NULL, propInstSub INTEGER NOT NULL, propInstObj INTEGER NOT NULL, PRIMARY KEY (propID, propInstSub, propInstObj));

CREATE INDEX PropertyInstReasonedPropID_IDX
ON PropertyInstReasoned(propID);

CREATE INDEX PropertyInstReasonedPropInstSub_IDX
ON PropertyInstReasoned(propInstSub);

CREATE TABLE Restriction (restID INTEGER NOT NULL, propID INTEGER NOT NULL, restSub INTEGER NOT NULL, restObj INTEGER NOT NULL, restType CHAR(4) NOT NULL, --some, any, allValues restCardMin INTEGER, restCardMax INTEGER, PRIMARY KEY (restID), FOREIGN KEY (propID) REFERENCES term (termID) ON DELETE CASCADE, FOREIGN KEY (restObj) REFERENCES term (termID) ON DELETE CASCADE, FOREIGN KEY (restSub) REFERENCES term (termID) ON DELETE CASCADE);

CREATE TABLE RestrictionReasoned (propID INTEGER NOT NULL, restSub INTEGER NOT NULL, restObj INTEGER NOT NULL, restType CHAR(4) NOT NULL, --some, any, allValues restCardMin INTEGER, restCardMax INTEGER, PRIMARY KEY (propID, restSub, restObj, restType));

CREATE INDEX RestrictionReasonedPropID_IDX
ON RestrictionReasoned(propID);

CREATE INDEX RestrictionReasonedRestSub_IDX
ON RestrictionReasoned(restSub);
A.3 XSLT

The following XSLT file is used for the transformation of the OWL file to a database readable SQL format. A detailed description of the XSLT transformation process can be found in section 5.3.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:local="uri.local-functions"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#
    xmlns:owl="http://www.w3.org/2002/07/owl#">

<xsl:output method="text"/>

<xsl:template name="tab">
    <xsl:for-each select="../../*[1]">
        #tab#
        <xsl:call-template name="tab"/>
    </xsl:for-each>
</xsl:template>

<xsl:template match="/">
    -- starting import
    CALL init_session;
    -- namespaces added via shell and sed
    -- sed_namespace_placeholder
    <xsl:apply-templates/>
    <xsl:call-template name="instances" select="/"/>
    CALL close_session;
</xsl:template>

<xsl:template match="owl:Ontology">
    -- create ontology
    CALL create_ontology(''<xsl:call-template name="URIreference"/>');
    <xsl:call-template name="OntologyProperty"/>
    <xsl:call-template name="AnnotationProperty"/>
    CALL stack_pop();
</xsl:template>

<xsl:template match="URIreference">
    <!-- generic template for ID, resource and about -->
    <xsl:value-of select="@rdf:about"/>
    <xsl:value-of select="@rdf:resource"/>
</xsl:template>

</xsl:stylesheet>
```
APPENDIX A.3. XSLT

<xsl:value-of select="@rdf:ID"/>
</xsl:template>

<xsl:template name="AnnotationProperty">
  <xsl:for-each select="owl:versionInfo | rdfs:label | rdfs:comment | rdfs:seeAlso | rdfs:isDefinedBy">
    CALL create_prop_inst_sis(''<xsl:value-of select="name()"/>'', stack_last_id(), '<xsl:value-of select="."/>'');
  </xsl:for-each>
</xsl:template>

<xsl:template name="OntologyProperty">
  <xsl:for-each select="owl:imports | owl:priorVersion | owl:backwardCompatibleWith | owl:incompatibleWith">
    CALL create_prop_inst_sis(''<xsl:value-of select="name()"/>'', stack_last_id(), '<xsl:value-of select="."/>'');
  </xsl:for-each>
</xsl:template>

<xsl:template name="ObjectProperty">
  <xsl:for-each select="owl:equivalentProperty | owl:inverseOf | owl:differentFrom | owl:sameAs | rdfs:subPropertyOf">
    CALL create_prop_inst_sis(''<xsl:value-of select="name()"/>'', stack_last_id(), '<xsl:call-template name="URIreference"/>');
  </xsl:for-each>
</xsl:template>

<xsl:template name="fact">
  <xsl:call-template name="individual"/>
</xsl:template>

<xsl:template name="individual">
  <xsl:call-template name="description"/>
</xsl:template>

<xsl:template name="descriptionClass">
  <xsl:for-each select=".">
    <xsl:call-template name="axiomsClass"/>
    <xsl:call-template name="restriction"/>
    <xsl:call-template name="complementOf"/>
    <xsl:call-template name="classCombination"/>
    <xsl:call-template name="oneOf"/>
  </xsl:for-each>
</xsl:template>
APPENDIX A.3. XSLT

<xsl:template name="class">
   --- class creation
   <xsl:call-template name="URIreference"/>
   parent: <xsl:value-of select="name(.)"/>
   CALL create_term(''<xsl:call-template name="URIreference"/>'',
   <xsl:choose>
      <xsl:when test=".'/*
     'class'
      </xsl:when>
      <xsl:otherwise>
        'uri'
      </xsl:otherwise>
   </xsl:choose>);
   <xsl:call-template name="AnnotationProperty"/>
   <xsl:call-template name="ObjectProperty"/>
   <xsl:call-template name="descriptionClass"/>
   -- class creation finished <xsl:call-template name="URIreference"/>
   parent: <xsl:value-of select="name(.)"/>
</xsl:template>

<xsl:template name="checkForClassCreation">
<xsl:choose>
   <xsl:when test="name()='owl:deprecatedClass'">
      <xsl:call-template name="createDeprecatedClass"/>
   </xsl:when>
   <xsl:when test="name()='owl:Class'">
      <xsl:call-template name="class"/>
      CALL stack_pop();
   </xsl:when>
   <xsl:when test="name()='owl:Restriction'">
      <xsl:call-template name="class"/>
      CALL stack_pop();
   </xsl:when>
</xsl:choose>
</xsl:template>

<xsl:template match="owl:Class">
   <xsl:call-template name="class"/>
   CALL stack_pop();
</xsl:template>

<xsl:template name="createDeprecatedClass">
   <xsl:call-template name="class"/>
   <!-- create property declaring the class as deprecated -->

72 Wolfgang Gassler, Eva Zangerle
CALL create_prop_inst_sis('rdfs:subClassOf',
  stack_last_id(), 'deprecatedClass');
<!-- CALL stack_pop(); -->
<!-- pops the class defined above -->
CALL stack_pop();
</xsl:template>

<xsl:template match="owl:deprecatedClass">
  <xsl:call-template name="createDeprecatedClass"/>
</xsl:template>

  CALL create_term('<xsl:call-template name="URIreference"/>', 'property ');
  CALL create_prop_inst_sis('rdfs:type', stack_last_id()
    , '<xsl:value-of select="name()"/>');
  <xsl:call-template name="descriptionProperty"/>
  <xsl:call-template name="AnnotationProperty"/>
  <xsl:call-template name="ObjectProperty"/>
  CALL stack_pop();
</xsl:template>

<xsl:template name="descriptionProperty">
  <xsl:for-each select="rdfs:domain | rdfs:range | rdf:type">
    CALL create_prop_inst_sis('<xsl:value-of select="name()"/>
      ', stack_last_id()
      , '<xsl:call-template name="URIreference"/>');
  </xsl:for-each>
</xsl:template>

<xsl:template name="axiomsClass">
  <xsl:for-each select="owl:equivalentClass | owl:disjointWith | rdfs:subClassOf">
    <xsl:when test="name()"/>
    parent <xsl:value-of select="name(.)"/>
    <xsl:choose>
      <!-- checks if class is defined within tag or
       only uri is stated -->
      <xsl:when test="./*"/>
      <xsl:for-each select="./*">
        <xsl:call-template name="checkForClassCreation"/>
        CALL create_prop_inst_sii('<xsl:value-of select="name(.)"/>
          , stack_last_id(),
          stack_magic_popped_id());
      </xsl:for-each>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>
CALL create_prop_inst_sis('<xsl:value-of select="name()"/>', stack_last_id(), '<xsl:call-template name="URIreference"/>');
</xsl:otherwise>
</xsl:choose>

--- <xsl:value-of select="name()"/> finished, parent <xsl:value-of select="name(../.)"/>
</xsl:for-each>
</xsl:template>

<xsl:template name="restriction">
  <!-- to avoid empty restriction start/end comments, check if the current node is a restriction -->
  <xsl:if test="name()='owl: Restriction'"/>
  --- restriction started concerning <xsl:value-of select="name(../.)"/>
  <xsl:choose>
    <xsl:when test="owl:allValuesFrom | owl:someValuesFrom | owl:hasValue">
      <xsl:for-each select="owl:allValuesFrom | owl:someValuesFrom | owl:hasValue">
        <xsl:for-each select="/*/">
          <xsl:call-template name="checkForClassCreation"/>
        </xsl:for-each>
      </xsl:for-each>
      <xsl:choose>
        <!-- restriction value is a literal -->
        <xsl:when test="normalize-space(.)">
          <xsl:call-template name="restrictionContent"/>
        </xsl:when>
        <!-- restriction value is a reference -->
        <xsl:when test="@rdf:ID | @rdf:about | @rdf:resource | ./*[@rdf:ID][1] | ./*[@rdf:about][1] | ./*[@rdf:resource][1]">
          <xsl:call-template name="restrictionContent"/>
        </xsl:when>
        <!-- class is defined within tag -->
        <xsl:otherwise>
          CALL create_restriction_objid(
            <xsl:for-each select="../owl:onProperty/*[1]">
              <!-- property -->
              '<xsl:call-template name="URIreference"/>
            </xsl:for-each>
          stack_last_id(), stack_magic_popped_id(), '<xsl:value-of select="name()"/>
        <xsl:call-template name="restrictionContent"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:if-test>
</xsl:template>
template name="cardinality="/>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
<xsl:when>
<xsl:when test="owl:cardinality | owl:minCardinality | owl:maxCardinality">
CALL create_restriction(
<xsl:for-each select="owl:onProperty/*/1">
 -- property -->
'<xsl:call-template name="URIreference"/>
</xsl:for-each>
stack_last_id(), 'NULL', 'cardinality', <xsl:call-template name="cardinality"/>, 'uri');
</xsl:when>
</xsl:choose>
-- restriction ended <xsl:value-of select="name()"/>
</xsl:if>
</xsl:template>

<xsl:template name="restrictionContent">
CALL create_restriction(
<xsl:for-each select="../owl:onProperty/*/1">
 -- property -->
'<xsl:call-template name="URIreference"/>
</xsl:for-each>
<!-- create restriction with object, which can be either specified datatype (uri) or literal -->
stack_last_id(),
<xsl:choose>
<!-- id stated in hasValue and similar components -->
<xsl:when test="@rdf:ID | @rdf:about | @rdf:resource ">
'<xsl:call-template name="URIreference"/>
</xsl:when>
<!-- id is stated within contained element e.g. class -->
<xsl:when test="/.*">
<xsl:for-each select="./*[1]">
'<xsl:call-template name="URIreference"/>
</xsl:for-each>
</xsl:when>
<!-- hasValue and similar contain a literal -->
<xsl:otherwise>
'<xsl:value-of select="."/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
APPENDIX A.3. XSLT

'\n
<xsl:template name="cardinality">
  <xsl:choose>
    <!-- cardinality is given -->
    <xsl:when test="owl:cardinality">
      <xsl:value-of select="."/>
    </xsl:when>
    <!-- minCardinality is given -->
    <xsl:when test="owl:minCardinality">
      <xsl:value-of select="owl:minCardinality"/>
    </xsl:when>
    <!-- maxCardinality given too -->
    <xsl:when test="owl:maxCardinality">
      <xsl:value-of select="owl:maxCardinality"/>
    </xsl:when>
    <!-- no maxCardinality given , set it to NULL -->
    <xsl:otherwise>
      NULL
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template name="oneOf">
  <xsl:for-each select="owl:oneOf/*">
    CALL create_prop_inst_sis('owl:oneOf', stack_last_id(), '<xsl:call-template name="URIreference"/>
  </xsl:for-each>
</xsl:template>

<xsl:template name="complementOf">
  <xsl:for-each select="owl:complementOf">
    CALL create_prop_inst_sis('owl:complementOf', stack_last_id(), '<xsl:call-template name="URIreference"/>
  </xsl:for-each>
</xsl:template>
<!-- check whether complement class is specified via ID or defined within -->
<xsl:choose>
  <xsl:when test="/*">
    <xsl:apply-templates select="owl:Class"/>
    CALL create_prop_inst_sii('owl:complementOf', stack_last_id(), stack_magic_popped_id());
  </xsl:when>
  <xsl:otherwise>
    CALL create_prop_inst_sis('owl:complementOf', stack_last_id(), '<xsl:call-template name="URIreference"/'>
  </xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</xsl:template>

<xsl:template name="classCombination">
  <xsl:for-each select="owl:intersectionOf | owl:unionOf">
    <xsl:for-each select="/*">
      CALL create_prop_inst_sii('<xsl:value-of select="name(..)"/>', stack_last_id(), stack_magic_popped_id());
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

<xsl:template name="dataRange">
  <xsl:for-each select="owl:DataRange">
    <xsl:choose>
      <xsl:when test="/*">
        <xsl:for-each select="/*">
          CALL create_prop_inst_sis('owl:dataRange', stack_last_id(), '<xsl:call-template name="URIreference"/'>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
        <!-- create datarange with one type, which can be either specified datatype (uri) or literal -->
        CALL create_prop_inst_sis('owl:dataRange', stack_last_id(), '<xsl:call-template name="URIreference"/>'
        <xsl:value-of select="."/>')
      </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>
APPENDIX A.3. XSLT

</xsl:choose>
</xsl:for-each>
</xsl:template>

<xsl:template name="instances">
<xsl:for-each select="rdf:RDF//*[rdf:ID] | rdf:RDF//*[rdf:about]">
<xsl:choose>
<xsl:when test="contains(name(), 'owl')"/>
<xsl:when test="contains(name(), 'rdf')"/>
<xsl:otherwise>
CALL create_term('<xsl:call-template name="URIreference"/>','inst');
CALL create_prop_inst_sis('rdfs:subClassOf', stack_last_id(), '<xsl:value-of select="name()"/>', 'uri');
<xsl:for-each select="*">
<xsl:call-template name="instanceProperty"/>
</xsl:for-each>
CALL stack_pop();
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</xsl:template>

<xsl:template name="instanceProperty">
<xsl:choose>
<xsl:when test="contains(name(), 'owl')"/>
<xsl:when test="contains(name(), 'rdf')"/>
<xsl:otherwise>
CALL create_prop_inst_sis('<xsl:value-of select="name()"/>', stack_last_id(), 'uri');
<xsl:choose>
<xsl:when test="normalize-space(.)">'lit'</xsl:when>
<xsl:otherwise> 'uri'</xsl:otherwise>
</xsl:choose>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>
A.4 SQL Procedures - Documentation

In the following section, the signatures of the SQL/PL procedures used during the Meta-Inn import and reasoning process are listed. More information about usage of the procedures can be found in section 5.3.

-- Initializes the import session, including stack and tmp relation
CREATE PROCEDURE "INIT_SESSION" ()

-- Pops the last element of stack argTermTypeName
CREATE PROCEDURE "STACK_POP" ( IN argTermTypeName VARCHAR(10) )

-- Pops the last element of stack term
CREATE PROCEDURE STACK_POP ()

-- Pushes argTermID to the stack argTermName
CREATE PROCEDURE stack_push ( IN argTermID INTEGER, IN argTermName VARCHAR(10) )

-- Adds the namespace to the current import session
-- ns name of the namespace (e.g. owl, rdf...)
-- uri full qualified uri of the namespace
CREATE PROCEDURE add_namespace ( IN ns VARCHAR(24), IN uri VARCHAR(1024) )

-- Changes the term name of specified term
-- argTermID id of the term
-- argName new term name to which the name is changed
CREATE PROCEDURE CHANGE_TERM_NAME ( IN argTermID INTEGER, IN argName VARCHAR(1024) )

-- Changes the term type of specified term
-- argTermID id of the term
-- argTermTypeName new term type to which the type is changed
CREATE PROCEDURE CHANGE_TERM_TYPE ( IN argTermID INTEGER, IN argTermTypeName VARCHAR(10) )

-- Closes the session and clean the environment relations (stack, tmp)
CREATE PROCEDURE "CLOSE_SESSION" ()

-- Sets the current ontology in the current session
-- needed by some other functions and procedures to know on which ontology they should operate
CREATE PROCEDURE "SET_ONTOLOGY" ( IN argUri VARCHAR (1024) )
-- Creates a new ontology and set it to the current ontology in the session
-- If the specified argUri already exist, the respective ontology is used and
-- set to the current ontology in the session
CREATE PROCEDURE "CREATE_ONTOLOGY" ( IN argUri VARCHAR (1024) )
-- Creates a property instance
-- argPropID termID of the property
-- argSubID termID of the subject on which the property is created
-- argObjID termID of the object value of the property
CREATE PROCEDURE "CREATE_PROP_INST_III" ( IN argPropID INTEGER, IN argSubID INTEGER, IN argObjID INTEGER )
-- Creates a property instance
-- argPropName name of the property (fully qualified uri or ns prefix, resolved by using namespaces)
-- argSubID termID of the subject on which the property is created
-- argObjID termID of the object value of the property
CREATE PROCEDURE "CREATE_PROP_INST_SII" ( IN argPropName VARCHAR (1024), IN argSubID INTEGER, IN argObjID INTEGER )
-- Creates a property instance
-- argPropName name of the property (fully qualified uri or ns prefix, resolved by using namespaces)
-- argSubID termID of the subject on which the property is created
-- argObjName object value, which is resolved to the respective instance if possible,
-- otherwise a new term of type literal is created
-- argObjTermType name of the object’s term type, e.g. uri, class, ...
CREATE PROCEDURE "CREATE_PROP_INST_SIS" ( IN argPropName VARCHAR (1024), IN argSubID INTEGER, IN argObj VARCHAR (1024), IN argObjTermType VARCHAR (10) )
-- Creates a property instance
-- argPropName name of the property (fully qualified uri or ns prefix, resolved by using namespaces)
APPENDIX A.4. SQL PROCEDURES - DOCUMENTATION

-- argSubID termID of the subject on which the property is created
-- argObjName object value, which is resolved to the respective instance if possible,
-- otherwise a new term of type literal is created.
-- object's term type is set to uri by default.
CREATE PROCEDURE "CREATE_PROP_INST_SIS" ( IN argPropName VARCHAR(1024), IN argSubID INTEGER, IN argObj VARCHAR(1024) )

-- Creates a property instance
-- argPropName name of the property (fully qualified uri or ns prefix, resolved by using namespaces)
-- argSub subject name, which is resolved to the respective instance if possible,
-- otherwise a term of type uri is created.
-- To keep data consistent, it should be changed to e.g. class later
-- argObj object value, which is resolved to the respective instance if possible,
-- otherwise a new term of type literal is created.
-- argObjTermTypeName name of the object's term type, e.g. uri, class, ...
CREATE PROCEDURE "CREATE_PROP_INST" ( IN argPropName VARCHAR(1024), IN argSub VARCHAR(1024), IN argObj VARCHAR(1024), IN argObjTermTypeName VARCHAR(10) )

-- Creates a property instance
-- argPropName name of the property (fully qualified uri or ns prefix, resolved by using namespaces)
-- argSub subject name, which is resolved to the respective instance if possible,
-- otherwise a term of type uri is created.
-- To keep data consistent, it should be changed to e.g. class later
-- argObj object value, which is resolved to the respective instance if possible,
-- otherwise a new term of type literal is created.
-- object's term type is set to uri by default.
CREATE PROCEDURE CREATE_PROP_INST ( IN argPropName VARCHAR(1024), IN argSub VARCHAR(1024), IN argObj VARCHAR(1024) )

-- Creates a property instance
-- argPropName name of the property (fully qualified uri or ns prefix, resolved by using namespaces)
-- argSub subject name, which is resolved to the respective instance if possible,
-- otherwise a term of type uri is created.
-- To keep data consistent, it should be changed to e.g. class later
-- argObj object value, which is resolved to the respective instance if possible,
-- otherwise a new term of type literal is created.
-- argObjTermTypeName name of the object’s term type, e.g. uri, class, ...
CREATE PROCEDURE CREATE_PROP_INST_SSS (IN argPropName VARCHAR(1024), IN argSubj VARCHAR(1024), IN argObj VARCHAR(1024), IN argObjTermTypeName VARCHAR(10))

-- Creates a restriction
-- argPropName name of the restricted property (fully qualified uri or ns prefix, resolved by using namespaces)
-- argSubID subject id on which the restriction is set
-- argObjID termID of the object value
-- argRestType someValuesFrom, allValuesFrom, hasValue or cardinality
-- argCardMin minimum cardinality or NULL
-- argCardMax maximum cardinality or NULL
CREATE PROCEDURE "CREATE_RESTRICTION_OBJID" (IN argPropName VARCHAR(1024), IN argSubID INTEGER, IN argObjID INTEGER, IN argRestType VARCHAR(100), IN argCardMin INTEGER, IN argCardMax INTEGER)

-- Creates a restriction
-- argPropName name of the restricted property (fully qualified uri or ns prefix, resolved by using namespaces)
-- argSubID subject id on which the restriction is set
-- argObjName name of the object value, resolved if possible, otherwise term of type literal is created
-- argRestType someValuesFrom, allValuesFrom, hasValue or cardinality
-- argCardMin minimum cardinality or NULL
-- argCardMax maximum cardinality or NULL
-- argObjTypeName object’s type name (e.g. uri, class, ...) CREATE PROCEDURE "CREATE_RESTRICTION" (IN argPropName VARCHAR(1024), IN argSubID INTEGER, IN argObjName VARCHAR(1024), IN argRestType VARCHAR(100), IN argCardMin INTEGER, IN argCardMax INTEGER, IN argObjTypeName VARCHAR(10))

-- Creates a restriction
-- argPropName name of the restricted property (fully qualified uri or ns prefix, resolved by using namespaces)
CREATE PROCEDURE "CREATE_RESTRICTION" (IN argPropName VARCHAR(1024), IN argSubID INTEGER, IN argObjName VARCHAR(1024), IN argRestType VARCHAR(100), IN argCardMin INTEGER, IN argCardMax INTEGER)

-- Creates a term and pushes the termID of the created term to the term stack
-- argTermName name of the term (if no name is specified, a random unique name is used)
-- argTermTypeName type of the term (e.g. class, uri, ...)
CREATE PROCEDURE "CREATE_TERM" (IN argTermName VARCHAR(1024), IN argTermTypeName VARCHAR(10))

-- Inserts the bootstrap information containing the following information
-- basic owl terms
-- basic rdf(s) terms
-- definition of transitive and symmetric properties
CREATE PROCEDURE "CREATE_BOOTSTRAP" ()

-- Resets the database and removes all imported data
-- Attention: all data get lost!
CREATE PROCEDURE rambo ()

-- Starts the reasoning process
-- All rules are applied as long as no new facts can be derived
-- The operations are performed on the relations RestrictionReasoned and propertyInstReasoned
-- If the reasoned relations are empty, the basic knowledge from Restriction and propertyInst is copied before the reasoning process is started
CREATE PROCEDURE DO_REASONING ()

-- Removes all reasoned data and starts the reasoning process
-- it calls do_reasoning (see do_reasoning for further information)
CREATE PROCEDURE REFRESH_REASONING ()
A.5 User Defined Functions - Documentation

The following listing shows the signatures of the User Defined Functions programmed in SQL/PL, which are used during the import and reasoning process of the Meta-Inn Approach. The usage of these functions is described in section 5.3.

-- Returns the ontology id of the current session
-- can only used if the set or create_session
-- procedure was called
CREATE FUNCTION "GET_CURRENT_ONTOLOGY" ()
RETURNS INTEGER

-- Returns the term type id of the specified
-- argTermTypeName
-- if the term type is not found, it is created
CREATE FUNCTION "GET_CREATE_TERM_TYPE_ID" ( argTermTypeName VARCHAR(10) )
RETURNS TABLE ( termTypeId INTEGER )

-- Returns the term type id of the specified
-- argTermTypeName
CREATE FUNCTION "GET_TERM_TYPE_ID" ( argTermTypeName VARCHAR(10) )
RETURNS INTEGER

-- Returns the specified string without the leading #
-- character
CREATE FUNCTION REMOVE_POUND ( string VARCHAR(1024) )
RETURNS VARCHAR(1024)

-- Returns the termID of the specified term name
-- argTermName name of the term (must be a fully
-- qualified url or a local name)
-- argTermTypeName term type name (must exist in the
-- database)
CREATE FUNCTION GET_TERM_ID ( argTermName VARCHAR(1024) , argTermTypeName VARCHAR(10) )
RETURNS INTEGER

-- Try to resolve the uri and returns the fully
-- qualified path
-- leading # char is removed and namespaces are
-- resolved
-- namespaces and ontology id must exist in the
-- current session
CREATE FUNCTION RESOLVE_URI ( argUri VARCHAR(1024) )
RETURNS VARCHAR(1024)
-- Returns the termID of the specified term name
-- if the term or term type doesn’t exist, it is created
-- argTermName name of the term (is resolved if necessary)
-- argTermTypeName name of the term type (is created if necessary)
CREATE FUNCTION "GET_CREATE_TERM_ID" ( argTermName VARCHAR(1024), argTermTypeName VARCHAR(10) )
RETURNS TABLE ( termId INTEGER )

-- Returns a result set containing all property instances of the specified property and class
-- argPropName fully qualified name of the property
-- argClassName name of the class
CREATE FUNCTION get_property_instances ( argPropName VARCHAR(1024), argClassName VARCHAR(1024) )
RETURNS TABLE ( termName VARCHAR(1024), termID INTEGER )

-- Returns a random name with the specified prefix
-- the random function is realized by appending a unique sequence number
CREATE FUNCTION GET_RANDOM_NAME ( prefixType VARCHAR(1013) )
RETURNS VARCHAR(1024)

-- Returns the id of a specified stack position
-- argValue type of the stack (e.g. term)
-- argPosition position specified by an integer (e.g. -2 = second position)
CREATE FUNCTION "STACK_ID" ( argValue VARCHAR(10), argPosition INTEGER )
RETURNS INTEGER

-- Returns the id of a specified position of the term stack
-- argPosition position specified by an integer (e.g. -2 = second position)
CREATE FUNCTION "STACK_ID" ( argPosition INTEGER )
RETURNS INTEGER

-- Returns the last termID of the specified stack
-- argTermTypeName type of the stack (e.g. term)
CREATE FUNCTION "STACK_LAST_ID" ( argTermTypeName VARCHAR(10) )
RETURNS INTEGER
APPENDIX A.5. USER DEFINED FUNCTIONS - DOCUMENTATION

-- Returns the last termID of the term stack
CREATE FUNCTION "STACK_LAST_ID" ()
  RETURNS INTEGER

-- Returns the last popped if of the specified stack
-- argTermTypeName type of the stack (e.g. term)
CREATE FUNCTION "STACK_MAGIC_POPPED_ID" (argTermTypeName VARCHAR(10))
  RETURNS INTEGER

-- Return the last popped if of the term stack
CREATE FUNCTION "STACK_MAGIC_POPPED_ID" ()
  RETURNS INTEGER

-- Returns a result set, containing all transitive reasoned facts
-- realized by a recursive SQL query on the relation propertyinstreasoned
-- only properties of the type TransitiveProperty are used for the reasoning
CREATE FUNCTION REASON_TRANSITIVE_PROPS ()
  RETURNS TABLE ( propid INTEGER, propinstsub INTEGER, propinstobj INTEGER, iteration INTEGER )

-- Returns a result set, containing all symmetric reasoned property instantiations
-- all property instantiations of symmetric properties are returned with swapped subject and object values
CREATE FUNCTION REASON_SYMMETRIC_PROPS ()
  RETURNS TABLE ( propid INTEGER, propinstsub INTEGER, propinstobj INTEGER )

-- Returns a result set, containing all inverse reasoned property instantiations
CREATE FUNCTION REASON_INVERSE_PROPS ()
  RETURNS TABLE ( propid INTEGER, propinstsub INTEGER, propinstobj INTEGER )

-- Returns a result set, containing all restriction inherited to subclasses
CREATE FUNCTION REASON_RESTRICTION_SUB ()
  RETURNS TABLE ( propid INTEGER, restSub INTEGER, restObj INTEGER, restType CHAR(4), restCardMin INTEGER, restCardMax INTEGER )

-- Returns a result set, containing all properties inherited to subclasses
CREATE FUNCTION REASON_PROPERTY_SUB ()
RETURNS TABLE ( propid INTEGER, propinstsub INTEGER, propinstobj INTEGER )

-- Returns a result set, containing all properties and restrictions of the specified class
-- the facts are based on the unreasoned data
-- argTermName name of the class, which should be described
CREATE FUNCTION DESCRIBE_CLASS ( argTermName VARCHAR (1024) )
RETURNS TABLE(Class VARCHAR(1024), Property VARCHAR(1024), PropertyValue VARCHAR(1024), RestrictionType CHAR(4), minCardinality INTEGER, maxCardinality INTEGER)

-- Returns a result set, containing all properties and restrictions of the specified class
-- the facts are also contain derived facts of the reasoning process
-- argTermName name of the class, which should be described
CREATE FUNCTION DESCRIBE_REASONED_CLASS ( argTermName VARCHAR (1024) )
RETURNS TABLE(Class VARCHAR(1024), Property VARCHAR(1024), PropertyValue VARCHAR(1024), RestrictionType CHAR(4), minCardinality INTEGER, maxCardinality INTEGER)
APPENDIX A.6. EXAMPLE ONTOLOGY

A.6 Example Ontology

The following OWL file contains an example ontology, which is used to explain the main concepts of this thesis.

```xml
<?xml version="1.0"?>
<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:owl="http://www.w3.org/2002/07/owl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xmlns="http://dbis-informatik.uibk.ac.at/owl/
 animal_ontology.owl#"
 xml:base="http://dbis-informatik.uibk.ac.at/owl/
 animal_ontology.owl" >

<owl:Ontology rdf:about="AnimalExampleOntology">
 <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal ontology</rdfs:comment>
 <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string">animal_ontology</rdfs:label>
 <owl:versionInfo rdf:datatype="http://www.w3.org/2001/XMLSchema#string">1.0</owl:versionInfo>
</owl:Ontology>

<!-- ############### Habitats ############### -->

<owl:Class rdf:ID="Land">
 <rdfs:subClassOf>
  <owl:Class rdf:ID="Habitat"/>
 </rdfs:subClassOf>
 <owl:disjointWith rdf:resource="#Water"/>
</owl:Class>

<owl:Class rdf:ID="Water">
 <owl:complementOf rdf:resource="#Land"/>
 <rdfs:subClassOf rdf:resource="#Habitat"/>
 <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">habitat for fish</rdfs:comment>
</owl:Class>

<!-- ############### Animals ############### -->

<owl:Class rdf:ID="Ocean">
 <rdfs:subClassOf rdf:resource="#Water"/>
</owl:Class>

<!-- ##################### Animals ##################### -->
```

Wolfgang Gassler, Eva Zangerle
APPENDIX A.6. EXAMPLE ONTOLOGY

<owl:Class rdf:about="#Animal">
  <rdfs:comment datatype="http://www.w3.org/2001/XMLSchema#string">Superclass for all animals</rdfs:comment>
  <owl:versionInfo datatype="http://www.w3.org/2001/XMLSchema#string">1.0</owl:versionInfo>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="livesIn"/>
      </owl:onProperty>
      <owl:minCardinality>1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#Mammals">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <owl:disjointWith rdf:resource="#Fish"/>
</owl:Class>

<owl:Class rdf:ID="Dolphin">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:hasValue datatype="http://www.w3.org/2001/XMLSchema#boolean">true</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Mammals"/>
  <rdfs:comment datatype="http://www.w3.org/2001/XMLSchema#string">Dolphin is mammal and can swim</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="Fish">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#Water"/>
        </owl:Restriction>
      </owl:Class>
      <owl:ObjectProperty rdf:ID="livesIn"/>
    </owl:equivalentClass>
  </owl:Class>
</owl:Class>
APPENDIX A.6. EXAMPLE ONTOLOGY

```xml
<owl:Restriction>
  <owl:onProperty>
    <owl:FunctionalProperty rdf:about="#canSwim"/>
  </owl:onProperty>
</owl:Restriction>
<owl:Class rdf:about="#Animal"/>
<owl:intersectionOf>
  <owl:Class rdf:about="#Mammals"/>
</owl:Class>

<owl:Class rdf:ID="Clownfish">
  <rdfs:subClassOf rdf:resource="#Fish"/>
  <owl:Restriction>
    <owl:onProperty>
      <owl:FunctionalProperty rdf:about="#livesIn"/>
    </owl:onProperty>
    <owl:allValuesFrom rdf:resource="#Ocean"/>
  </owl:Restriction>
</owl:Class>

<owl:Class rdf:ID="Shark">
  <rdfs:subClassOf rdf:resource="#Fish"/>
  <owl:Restriction>
    <owl:onProperty>
      <owl:ObjectProperty rdf:about="#isFoodOf"/>
    </owl:onProperty>
    <owl:allValuesFrom rdf:resource="#Shark"/>
  </owl:Restriction>
</owl:Class>

<owl:Class rdf:ID="AnimalInOcean">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:allValuesFrom rdf:resource="#Ocean"/>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#livesIn"/>
          </owl:onProperty>
        </owl:Restriction>
      </owl:Class>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```
<owl:onProperty>
</owl:Restriction>
<owl:Class rdf:about="#Animal"/>
<owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>

<!-- ############### Food ############### -->

<owl:Class rdf:ID="AnimalFood">
<owl:unionOf rdf:parseType="Collection">
<owl:Class rdf:ID="#FoodInWater"/>
<owl:Class rdf:ID="#FoodOnLand"/>
</owl:unionOf>
</owl:Class>

<!-- ############### Properties ############### -->

<owl:FunctionalProperty rdf:ID="canSwim">
<rdfs:domain rdf:resource="#Animal"/>
<rdfs:range rdf:resource="#Animal"/>
</owl:FunctionalProperty>

<owl:ObjectProperty rdf:ID="livesIn">
<rdfs:domain rdf:resource="#Animal"/>
<rdfs:range rdf:resource="#Habitat"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isHabitatOf">
<rdfs:range rdf:resource="#Animal"/>
<rdfs:domain rdf:resource="#Habitat"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isFoodOf">
<rdfs:range rdf:resource="#Animal"/>
<rdfs:domain rdf:resource="#Animal"/>
</owl:ObjectProperty>

<!-- ############### Instances ############### -->

<Ocean rdf:ID="Pacific"/>
<Shark rdf:ID="TheWhiteShark"/>
APPENDIX A.6. EXAMPLE ONTOLOGY

<Shark rdf:resource="#Pacific"/>

<Dolphin rdf:ID="Flipper">
  <livesIn rdf:resource="#Pacific"/>
</Dolphin>

<Clownfish rdf:ID="Marlin">
  <livesIn rdf:resource="#Pacific"/>
</Clownfish>

<Clownfish rdf:ID="Nemo">
  <livesIn rdf:resource="#Pacific"/>
</Clownfish>

</rdf:RDF>
Bibliography


[dBEF+06] Jos de Bruijn, Marc Ehrig, Cristina Feier, Francisco Martin-Requierta, Francois Scharffe, and Moritz Weiten. Semantic Web Technologies, chapter Ontology Mediation, Merging
APPENDIX BIBLIOGRAPHY


[MMP06] Jing Mei, Li Ma, and Yue Pan. Ontology Query Answering on Databases. In Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika,
APPENDIX BIBLIOGRAPHY


